Citation for published version:
Bilovich, A 2006, *Detecting individual differences in beliefs through language use*. Computer Science Technical Reports, no. CSBU-2006-22, Department of Computer Science, University of Bath.

Publication date:
2006

Link to publication

©The Author December 2006

University of Bath

Alternative formats
If you require this document in an alternative format, please contact:
openaccess@bath.ac.uk

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

Department of Computer Science

Technical Report

Undergraduate Dissertation: Detecting Individual Differences in Beliefs Through Language Use

Avri Bilovich

Technical Report 2006-22 **December 2006**

ISSN 1740-9497

Copyright ©December 2006 by the authors.

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Detecting Individual Differences in Beliefs Through Language Use

Avri Bilovich
BSc (Hons) in Computer Science
University of Bath,
Department of Computer Science
Bath BA2 7AY, UK
cs3aab@bath.ac.uk

May 5, 2006

Detecting Individual Differences in Beliefs Through Language Use

submitted by Avri Bilovich

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Avri Bilovich

Abstract

Individual differences in beliefs have up to now been identified primarily by questioning people or observing them in a series of tasks such as stimuli classification. The Implicit Association Test, which is a very useful tool for identifying these implicit beliefs uses exactly this approach. This project aims to provide an automatic tool which will identify individual differences in beliefs through text analysis. This will be very useful as research in individual beliefs will not have to rely on data gathered directly from people which is very time consuming, but on data gathered from written texts which is much more accessible. This paper describes an automatic way to detect individual difference in beliefs through language use, using semantic space theories. After reviewing the underlying theories necessary to understand the full scope of the project, a platform for detecting individual differences in beliefs (written in Lisp) is presented. The validation of the program shows that although the results can be optimised further individual differences in beliefs are detected. Finally the tool is used to explore the evolution of some beliefs as extracted from the Bible, the works of William Shakespeare and the British National Corpus (which represents common beliefs held today).

Contents

List of Figures

Acknowledgments

First and foremost I would like to thank Dr J. Bryson, my project supervisor for all the help and support provided to me during the course of this research, and my years at Bath University. This project could not have been completed without the help of Dr W. Lowe whose work, and that of the Implicit project, inspired this research. Mr Tim Francis and the Bath University Computer Services were a big help in getting hold of the data (the BNC) without which most of this research would have been near impossible. I am also grateful for my friends and colleagues who have proof-read parts of this thesis and provided me with valuable input, in particular Ms Maayan Navellou, Ms Amy Marshall and Mr Adam Poolman. I would also like to thank my family and friends for making my life easy and relaxed during this research period. In addition, I would like to thank all the academics who have done research in this field and have provided me with the building blocks for the work presented here. The different freely available online resources for the language Lisp have also been invaluable for the completion of this project. The CCVISU tool freely available online (http://mtc.epfl.ch/∼beyer/CCVisu/) was also very helpful in order to provide a force directed visualisation of word similarities and was used in order to produces the figures 4.1, 4.2 and 4.3. Finally, I would like to thank you, the reader, for taking an interest in this research project.

Chapter 1

Psychological Background

1.1 General Introduction

Individual differences in beliefs are often referred to in psychology as 'automatic attitudes'. These are usually unconscious attitudes which are moderated by the social environment in which one evolves. The Implicit project (Implicit-Project, 1998) explores these differences using the Implicit Association Test (Greenwald et al., 1998). Studies done using the IAT have shown many interesting results about implicit associations and automatic attitudes (Banaji and Greenwald, 1994; Greenwald and Nosek, 2001; Mitchell et al., 2003). As this is a test, it requires subjects to explicitly take part in it, in order to gather results about their beliefs. In this project we are proposing to further analyse the differences identified by the IAT, and to do so from a different angle. We will explore the identification and the implicit transmission of these mental biases via language use. This will make it much easier to gather information about latent cultural beliefs in the general population, by analysing a corpus of texts representing how language is used today. Given enough text written by a single person, it will also be possible to analyse his (or her) implicit beliefs, without direct interaction with the person.

Before presenting the way in which we will identify these differences in beliefs, we have to give some background about the psychological theories that underlie our work. This will include a more detailed description of the IAT and the theories that influenced it. We will also devote a section to the description of some of the findings of the IAT studies, and the results that we will try to replicate using our technique. Finally, we will present the technical theoretical background for our work: semantic space theory and will finish by reviewing the specific semantic space model that we will implement in our program.

1.2 Psychological background

In order to understand the validation of our program i.e. why we can claim that it is indeed able to extract latent cultural beliefs, one must have basic knowledge of some psychological theories. These include spreading activation theories, compound cue theories in addition to the semantic and mediated priming phenomenon.

1.2.1 Spreading activation theory

Spreading activation theory describes how knowledge is represented and used in the brain (Anderson, 1983). In this model, knowledge is represented as a undirected graph, or a network, where the vertices are concepts in long-term memory and the edges are associative pathways between these concepts. When these concepts, known more formally as *cognitive units* (Anderson, 1983) are activated, activation spreads along the associative pathways to neighbouring nodes. Therefore when one concept is activated by a stimulus, similar or closely related areas of memory are also activated to a lesser extent. This spreading of activation serves to make these related concepts available for further cognitive processing (Balota and Lorch Jr., 1986). In simpler terms, this means that when presented with a stimulus, for example the word *tiger* or an image of a tiger, all our knowledge closely related to the concept of a tiger that we hold, is activated. This means that it then becomes easier for us to reason about these related concepts. Thinking about tigers facilitates our thinking about lions, animals in general etc... The less related a concept is with our stimulus, the further away it is in the memory network and the less it is activated. This is described in more detail in Anderson (1983).

Spreading activation theories account for many memory and knowledge related processes (Sharifian and Samani, 1997; Anderson, 1983). We will now present the *mediated priming* process. Mediated priming and many other of these processes can also be explained by compound cue theories which we will describe shortly and by semantic space theories which is the theoretical basis of our program.

1.2.2 Priming and Mediated Priming

Priming, or associative priming refers to the facilitation in accessing information when associated items are present (Sharifian and Samani, 1997; Anderson, 1983). This manifests itself by the fact that words are recognised faster when they appear in the context of associated words than when they appear adjacent to un-associated words. Thus, the word *mouse* will be recognised faster when preceded by the word *cheese* than by the word *shirt*. This has been observed experimentally (Balota and Lorch Jr., 1986) and explained by different memory and knowledge theories: spreading activation theories (Anderson, 1983), compound cue theories (Ratcliff and McKoon, 1988) and semantic space theories (Lowe and McDonald, 2000).

When a word **A** is associated, and therefore facilitated, by a word **B** we say that **B** is a *prime* for **A**. When studying **A**, we refer to it as a *target*. Most experiments for observing priming effects rely on the measure of reaction time (RT) of the subject, when given a visual stimuli (Balota and Lorch Jr., 1986). The subject is asked to react to the visual stimuli by pressing a specific key on a keyboard which is dependent on the stimuli. It is shown the mean RT when presented with a target word (e.g. tiger) is shorter when it is precede with a prime (e.g. stripes) than with an unrelated word (e.g. shirt). This priming phenomenon is also called direct priming, as opposed to mediated priming. A mediated prime to a word **A**, is a word **C** such that **A** and **C** share a common prime **B**. For example, the words *lion* and *stripes* are mediated primes, as they are connected by a common prime *tiger*. Mediated priming effects are observed via similar experiments as direct priming (Balota and Lorch Jr., 1986; Lowe and McDonald, 2000). Spreading activation models account for mediated priming by allowing for recursive activation: when a node is activated in the network, the activation spreads to the neighbouring nodes and from each of these nodes it spreads recursively (Anderson, 1983). Thus, when the node containing our initial target is activated, its primes are activated and when each prime is activated, the related primes are then also (more weakly) activated. We will now look at another set of influential theories, which also provide explanations for the mediated priming phenomenon among other processes: compound cue theories.

1.2.3 Compound Cue Theories

Compound cue theories are more recent than spreading activation theories, and are less intuitively clear (Ratcliff and McKoon, 1995, 1988). The main difference between the two theories is the way that knowledge is represented and retrieved. Spreading activation theories organise information in a long-term memory network and when the memory system is presented with an item, a specific node is activated, and activation spread to nearby (i.e. related) nodes making them potentially more available to subsequent processes. Compound cue theories assume that stimuli presented closely together form compounds in short-term memory. These compounds are subsequently matched against information in long-term memory by a passive process, which returns a value of familiarity for each compound. Thus, in spreading activation theories, recognition and processing of information is done by spreading activation in the long-term memory network whereas in compound cue theories this is a consequence of the combination of items in short-term memory with only limited long term memory participation (Ratcliff and McKoon, 1995).

Priming and mediated priming are both accounted for in compound cue theories (Ratcliff and McKoon, 1988, 1992). The facilitation phenomenon that is observed is explained by compound cue theories by high familiarity values for the compounds. An item that forms into a high familiarity valued compound is facilitated. The familiarity value depends on the task, and therefore is computed differently when dealing with recognition, lexical decisions, etc. as described by Ratcliff and McKoon (1992).

Up to now, comparisons between spreading activation theories and compound cue theories have not been able to validate one theory more than another on priming tasks (Beer and Diehl, 2001). The facilitation observed in word associations (i.e. priming and mediated priming), which are explained by both these psychological theories, have influenced many studies into human cognition. Most noticeably, these effects have been used in detecting individual differences (Greenwald et al., 1998). We will now present some of these studies, before describing the main supporting theory behind our work, the semantic space theory, and how we wish to replicate these studies automatically (i.e. with no need of running direct tests on subjects).

1.2.4 Detecting Individual Differences: the Implicit Association Test

The Implicit Association Test (IAT) as proposed by Greenwald et al. (1998), has been the basis of numerous projects exploring individual differences in beliefs. These projects have shown many interesting conclusions about implicit associations and automatic attitudes (Banaji and Greenwald, 1994; Greenwald and Nosek, 2001; Mitchell et al., 2003). For example stereotyping, prejudicial attitudes and discriminatory behaviour have been shown to have an unconscious basis (Banaji and Greenwald, 1994), which leads to new theories about how to alter them (by concentrating on the unconscious component of the behaviour instead of the conscious one).

The IAT is used to measure *relative strength of associations* between pairs of concepts. For example, in using the pair of associations 'male — female' and 'mathematics — art' we may discover that the association 'male — mathematics' is stronger than 'female — mathematics' and that therefore there is a clear gender — mathematics bias in the tested population (Lemm and Banaji, 1999). The tests itself consists of a simple task of sorting visual stimuli (words or pictures) into two categories. For example, if we wish to try and correlate the 'gender — math' stereotype, than we will use words or images which are related to mathematics, arts and some gender specific information (e.g. pictures of men and male names and the same with female names and pictures). The subject is seated in front of a screen and in a first run, is asked to categorise as quickly as possible each stimuli appearing on the screen into two categories. Any stimuli relating to 'mathematics' is to be put into the 'male' category (inversely the 'arts' related words and images are to be aligned with the 'female' category). A second run asks the subject to do the inverse task i.e. sorting the same sets of stimuli into the other categories 'art' with 'male' and 'mathematics' with 'female'. It is also possible to present the subject with two additional tasks, reversing the roles of the stimuli and the categories: sorting words or images relating to 'male' (respectively 'female') into the 'arts' and 'mathematics' categories. The IAT works under the assumption that it will be easier to sort words into categories that we find strongly associated with them. For example if a person is biased against black people, it is assumed that associating the picture of a black person with the category 'unpleasant' will be easier than associating

the same stimuli with the category 'pleasant'. In this context, 'easy' refers to a quicker reaction time and a low error rate observed during the sorting tasks presented to the subjects.

1.2.5 Detecting individual beliefs through language use

In contrast to this theory, we speculate that these associations are also transmitted implicitly by individuals through language. Thus by analysing the way people use language, we will be able to collect automatically the data that the IAT acquires through direct experimentation, which requires the active participation of individuals. In addition, it is possible that some of the associations identified by the IAT are in fact a product of an evolved culture which is influenced amongst other things by texts we read. This would show that literature propagates (implicitly) biases and helps to build a 'cultural identity', i.e. a relatively common set of beliefs which are shared by a significant part of the population. Direct exchange of ideas and opinions through discussion or written text is an effect commonly observed in everyday life. A more interesting question to ask oneself, is if there is any unconscious information that is indirectly transmitted in any form of verbal or written communication, even one not aimed at persuading others to a particular opinion. Newspapers, for example, relate information differently, depending mainly on their political views. The beliefs of these newspapers indeed mediates the beliefs of the population that is exposed to it (Dunn et al., 2005). As we will see, our program is able to extract individual beliefs that are implicitly conveyed in this fashion, and thus enables us to better analyse such phenomenon. The theoretical background of this program is the semantic space theory and the derived models.

In the context of this project, we are looking for automated ways of analysing the use of language. Semantic Space Models are used to analyse different properties of words or documents. For example the Latent Semantic Analysis (LSA) model that we will describe later has applications in document indexing and retrieval (Deerwester et al., 1990), and can be used for representation of the knowledge gained from a document (Landauer and Dumais, 1997). Landauer (2002) describes many other applications of this model. We will now present an introduction to Semantic Space Models in general and a few specific models, including LSA. We will then review the usefulness of these models in relation to our project.

1.3 Semantic Space Models

1.3.1 Semantic Space Theory

The Semantic Space theory is based on the assumption that the context in which words are used gives us some information about them (Lowe, 2001). There are two types of information that can thus be gathered about a word. Firstly, the lexical surrounding of a word gives us syntactic information about it. Secondly,

and in our case more importantly, we can even know about the semantics of a word in relation to other words. Indeed, the intuitive postulate here is that if a word has a similar statistical distribution as another word, then it is more closely related to it than to a word with a completely different statistical distribution. That is to say: we use closely related words similarly. In fact, some theories state that it is in this way that we accumulate knowledge of new words and their meaning (Landauer and Dumais, 1997; Landauer, 2002). While we are reading, when we are faced with an unknown word, we look at the context and associate that word to another one of which we know the meaning. We choose which word to associate it with depending on the context: a word will be associate with one which has a similar context. There are even statistical replacement test (Finch, 1993) which test the meaningfulness of interchanging similar words within their exact context. For example, if we are faced with a word **A** which we do not understand, we can try and replace it with a word **B** which is known to us and occurs in a similar context as **A**. The statistical replacement test can help us asses the validity of such replacement, and thus indirectly, give us an idea of the similarity between the meaning of **A** and **B**.

Semantic space models represent this similarity between words by mapping them into an Euclidean n-dimensional space. The axis of this space are context words: words that serve as a context to those that we are studying. Each of the studied words, called 'target words', is represented as a vector in that space. The vector for the word is in fact a vector of word co-occurrence statistics and can be constructed automatically by a simple program. In order to compare words, we first of all need them to be in the same semantic space[1]. We can then use either the Euclidean distance between the two points or the cosine of the angles between them as a measurement of similarity. The square Euclidean distance between to vectors $\mathbf{v}$ and $\mathbf{w}$ in D dimensions, written as $\|\mathbf{v}-\mathbf{w}\|^2$, is related to the cosine μ_{vw} of the angle between them as follows:

$$\begin{aligned} \|\mathbf{v}-\mathbf{w}\|^2 &= \sum_{i=1}^{D}(v_i-w_i)^2 && (1.1)\\ &= \|\mathbf{v}\|^2+\|\mathbf{w}\|^2-2\frac{\mathbf{vw}}{||\mathbf{v}||||\mathbf{w}||} && (1.2)\\ &= \|\mathbf{v}\|^2+\|\mathbf{w}\|^2-2\mu_{vw} && (1.3) \end{aligned}$$

where$\|\mathbf{w}\|^2 = \sum_i^D w_i^2$ is the square of the vector's length (Lowe, 2000, page 59). The advantage of using the cosine as a similarity measure is that it's range is [-1;1] and thus any arbitrary scaling factor introduced by the choice of the context words (also called the *basis* of the semantic space) is removed.

Different models use different methods for choosing the basis of the space and for measuring the similarity between words. We will now look at specific examples of Semantic Space Models, and what we can take from them into our current research. Let us start by looking at one of the most influential theories,

[1]Fodor and Lepor (1999) have a more detailed discussion of this, especially pages 12 onwards.

which has many applications in a wide range on domains, from information retrieval to automatic essay grading: Latent Semantic Analysis.

1.3.2 Latent Semantic Analysis (LSA)

Latent Semantical Analysis (Landauer and Dumais, 1997) is probably the most well known and influential model based on semantic spaces. Indeed, it has been used in a wide range of practical as well as theoretical domains (Landauer, 2002) from search algorithms and automatic essay grading to learning theories (Landauer and Dumais, 1997). As the other semantic space models, LSA is loosely based on gathering vectors of statistical word co-occurrences, given a text upon which to base the analysis. It is important to note that the similarity between words, as calculated via LSA is not simply based on the co-occurrence frequencies, but on a more powerful mathematical analysis method.

LSA starts by representing the given text as a matrix $\mathbf{M}$. Each row represent a unique word we are interested in. Each column stands for a specific context such as a paragraph, or a text passage. The matrix is first transformed using a normalisation function that weights each word by its importance in the particular context. Singular value decomposition (SVD) is then applied to the matrix, thus decomposing it into the product of three other matrices.

$$\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

The matrices $\mathbf{U}$ and $\mathbf{V}$ describe the rows and columns of $\mathbf{M}$ with respect to the base vectors associated with the singular values. The matrix $\mathbf{\Sigma}$ is a diagonal matrix that contains the scaling values (also called singular values) that enable us to reconstruct the original matrix. SVD is a well studied mathematical approach, and we are assured that any rectangular matrix can be decomposed as described above. In practice we can only compute SVD for matrices with only a few thousand dimensions due to computation time. This calculation can be done for example using the DGESVD subroutine in the LAPACK (Linear Algebra PACKage) software library (LAPACK, 2000) Once the matrix has been decomposed in this way, we reduce its dimensionality (i.e. the number of rows and columns). LSA theory (Landauer et al., 1998) assumes that the choice of dimensionality in which we define a word has great importance. By reducing the dimensionality of the words' representation, we approximate better human cognitive relations between them. It is important to identify the optimal dimensionality for the representation in order to get the most meaningful results. Indeed on testing LSA on the TOEFL test, Landauer and Dumais (1997) found that the chosen dimensionality for the LSA was crucial in the results. TOEFL is the Test Of English as a Foreign Language, and is used to check that students from non-English speaking countries applying to study in England or America have a good understanding of the language. LSA was tested on the synonym portion of the test, and scored around 64% with an optimal dimensionality. Other choices for dimensionality, either greater or less than the optimal one gave poor scores on the test (Landauer and Dumais, 1997; Levy and Bullinaria,

2001). It is interesting to note the fact that as Levy and Bullinaria (2001, page 9) state, "...dimensionality reduction is only useful in some cases, but only in certain ways of using the co-occurrence statistics". Therefore we must analyse the model that we will use and find out if indeed dimensionality reduction is essential to it, and what model of dimensionality reduction to use. We will explore this issue more in depth in section 1.4.

1.3.3 The Limitations of Semantic Space Models

In addition to the uses outlined in the above section, semantic space models have been shown to be able to model the process of mediated priming (Lowe and McDonald, 2000). Mediated priming is a well established phenomenon, which is explained by theories of semantic memory. Spreading activation theories predict that being presented with a prime word facilitates the pronunciation or lexical decision on a target word. For example the word 'tiger' facilitates 'stripes', as they are obviously related. Similarly 'lion' facilitates 'tiger', as they are also related. Therefore, to some degree, the word 'lion' also facilitates 'stripes' (de Groot, 1983). These effects were replicated using semantic spaces by Lowe and McDonald (2000), emphasising again the validity of these models.

Although semantic space models are very useful for many applications, such as those seen in the previous section about LSA, they are not perfect. Indeed Landauer and Dumais (1997) claims that LSA can emulate the acquisition of new vocabulary by people, and therefore extract meaning from words. French and Labiouse (2002) demonstrates the main flaws a semantic space based approach to this domain. The 'meaning' of a word, in semantic space terms is indeed its position in the specific space. This shows well that the meaning is in fact based on a particular basis, and by extension, when we compare words for similarity, we must place them in the same semantic space (Fodor and Lepor, 1999). Our usage of words is not restricted to using similar words in similar context. One can use a word in many unrelated contexts, for example when making an analogy. The analogy between an ice-cream and a medicine for example, relies on our own personal human experiences. If we have never had the experience of for example, been given ice-cream by our mother to make us feel better when we are sad, or by the doctor after we have had our tonsils removed, the analogy does not make any sense. It has been argued that a system using a semantic space model, will not be able to correctly 'understand' analogies, and when faced with a question such as "rate the similarity between an ice-cream and a medicine" it will give a very non-human response (French and Labiouse, 2002). In effect, critics say that the model lacks essential knowledge about the world that humans acquire through learning or direct experience. This is therefore a first example of a possible breakdown of the semantic space models to acquire human semantics. On the other hand it can also be argued that humans will also not be able to give expected reply to such a question if they would not have been influenced by the cultural knowledge that is continuously transmitted verbally or textually to them. A human not having directly experienced the removal of his tonsils and the ice cream given may still be able to understand the

analogy just by having read or having been told of someone having the similar experience. Given such data, a semantic space model can possibly be made to 'understand' such analogies. This would just be a replication of how humans understand these expressions using second hand experiences (i.e. experiences that have been communicated to them instead of experienced directly by them). It is interesting to look at this example in the light of the psychological theory of Constructivism (Piaget, 1950). This theory puts forward the fact that learning is done through direct interaction with the world. Programs relying on word co-occurrence counts, cannot replicate the whole learning process that takes place for every person while they constantly interact with the world. These programs can not hope to be able to have unique individual experiences, but they can still use the combined experience of many people, which is reflected in language, in order to 'learn' and be able to replicate humans in some understanding tasks (such as the synonym selection example presented earlier).

Research into the phenomenon of embodiment (Boroditsky and Ramscar, 2002) has shown that abstract knowledge is built analogically from more experience based knowledge. Embodiment, in simple terms is the influence of the body on the thought process. For example a sentence being interpreted differently depending on the activity that one is currently undertaking. Boroditsky and Ramscar (2002) has shown that the abstract domain of time is for example dependent on the domain of space, which is more experience based. Arguably one does not have experience of moving in time, as much as the experience of moving in space. Therefore the conclusions one draws when faced with concepts of movement in time, are directly related to analogies of movement in space. The study shows, via five experiments, that when a subject is reasoning about a certain type of movement in space, this directly affects his/her word understanding of movement in time. For example, a person waiting to travel on a train, and therefore thinking of himself as about to travel towards a destination, applies the same process to thinking about time, and sees himself as moving forwards in time. Semantic space based models cannot yet expect to model this cognitive behaviour. As we have already said, these models do not recognise that a word can be used in an unusual context, for an analogy for example, a similar concept is observed here. These models have no way of representing embodiment or the influence of experience based thoughts on more abstract ones. The only possible way to attempt to include such information within a semantic space model is by adding new sets of primes linking expressions such as 'travelling to' with other expressions regarding moving forwards in different contexts (such as time and space). As this is only a conjecture, at present semantic space model, gaining knowledge about the world only through word co-occurrence is not capable of differentiating abstract concepts from experienced based ones, let alone relating between them.

Another limitation in the semantic space model is the fact that, even with all the knowledge gained from word co-occurrences, a program cannot reason about words that it has not seen before. Humans can analyse sounds in words, or parts of words, and therefore decide if it is appropriate in a certain context. French (2000, page 5), has a very good example that emphasises the fact that

words are not atomic entities and can be analysed by the auditive or semantic components. The example *Flugly*, is all in all non-pleasant sounding, from an Anglo-American perspective, mainly due to its auditive components, such as the guttural 'ug' sound, and the word 'ugly' which is a part of it. Current systems that rely on word co-occurrence counts, are not sensitive to this information contained in words and miss therefore crucial information for reasoning about them.

All in all, semantic space models are very useful for many different Artificial Intelligence applications such as learning theory, document meaning analysis and comparison, and so on. Nevertheless, as we have highlighted in this section they are not without limitations. For the purposes of our study, analysing the use of language in order to identify individual differences in beliefs, a semantic space model certainly seems to be a very applicable solution. We will now review the main decisions made concerning the specific semantic space model used for the project's implementation and how we have tackled the problematic issues raised by the use of such a model.

1.4 Design decisions: problems and solutions

Using a semantic space based model in order to analyse differences in belief through the use of language, is motivated by the fact that such a model incorporates all the tools that we need in a simple and elegant fashion. In addition, the wide literature on semantic space models, that we have reviewed previously, shows that it is an area well established and has a solid mathematical basis (Lowe, 2001).

1.4.1 Design of the program

The program that we have developed reads through a text that we wish to analyse, and maps words in which we are interested into an appropriate semantic space. It then analyses the different meaning similarities in which we are interested, and outputs a quantitative measure of these similarities. Based on these quantitative measures, we can then deduce the bias and therefore the beliefs of the author of the text, in relation to the semantic space that we have chosen. Our efforts are mostly concentrated on the simplicity, and efficiency of the program: we try to ensure that we get the best results using mostly simple concepts in order to achieve this.

In order to have a meaningful result, we obviously need to choose what relations to look for. This choice will highly influence our semantic space, as the target words will be a direct consequence of that choice. If for example we are trying to detect a racial bias, towards black or white people, the target words will be words highly correlated with the colours black and white, and different designations for black or white people. In addition, we will map different words with positive or negative connotations to our semantic space, and will use them as a fixed comparative measure with out main target words. We will then be able

to quantify the similarity between the target words and the positive or negative ones, thus giving us a quantifiable measure of the text's bias towards one or the other concept. Indeed, if the similarity between words associated with white (and white people) and words with positive connotation is significantly greater than the similarity of black related words and the positive words, then we will be able to assume that the text is biased towards white people. The author of the text is therefore, consciously or unconsciously, biased towards white people. This test could be replicated with comparing the main target words with the ones with negative connotation, where we expect to see the inverse effect to the one observed in the previous test. This second test can give us another measure of the text bias, thus making the conclusions drawn from it more reliable. It is also possible that we detect a bias towards the colour white and not towards white people. The IAT uses visual stimuli or typical black and white American names in order to detect racial biases (Greenwald et al., 1998). Depending on the frequency of these names in the text that we will analyse, it might be possible to detect these implicit biases in this way if the first proposed experiment does not yield satisfactory results.

1.4.2 Data sparsity

The main problem that we will face follows from the problem of data sparsity. Zipf's law (Zipf, 1949) states that in a (syntactical and semantical correct) textual corpus, the frequency of any word is roughly proportional to the inverse of its rank in the frequency table. That is to say that the most frequent word in the text will occur about twice as often as the the second most frequent word, etc. A direct result of Zipf's law is that the first few most common words such as 'the', 'be', 'of', 'and' and 'a' represent most of the words that one will encounter in a text. In other words, a majority of words will occur very infrequently in a text. As the words we want to analyse are not part of the most common words (indeed it would be useless for us to look at the usage of the word 'the' for example) we have a data sparseness problem. This is the reason why we do not use vectors of co-occurrence counts directly, but we apply some form of normalisation function that factors out chance co-occurrences of words. This normalisation function is called a Lexical Association Function and is different for every model. We will use log odds-ratio described in Lowe and McDonald (2000) which was used for observing mediated priming in semantic space. We will compare this with a simpler method described in Levy and Bullinaria (2001), which divides each dimension value by the overall frequency of co-occurrence of the target word (slightly readjusted), in order to see which gives us the most relevant results.

The data sparseness problem is less problematic than it seems at first: the stimuli that we will use at first will be mostly derived from the IAT experiments as we wish to validate our program with regards to the identification of implicit beliefs. These stimuli will therefore be fairly common words. The only possible difficulty regarding the replication of these experiments, is the fact that the IAT uses many names in order to detect racial biases (Greenwald et al., 1998). It may

be the case that these names do not appear very frequently in the different texts that we will analyse (especially the Japanese and Korean names). If we would still want to replicate these exact results, with the same stimuli, we will have to possibly sum up all the co-occurrence values for the names in the different categories, and plot that as a single point in our semantic space, thus analysing all the names as a single element and not as different names. This technique may also be used when dealing with other categories having low frequency words that we wish to analyse.

1.4.3 Basis choice and dimensionality

Another very important choice that we are faced with is the choice of a reliable basis for our semantic space. Indeed, a bad choice for the basis, can highly influence our results and make them unusable. The choice of a basis will be decided by empirically testing the results given by the basis used by Lowe and McDonald (2000) and comparing them to the best performing one in Levy and Bullinaria (2001), which is simply the 8192 most frequent words in the *British National Corpus*. In addition, we will try to add the 147 most frequent words to the former basis of 536 words, in order to verify if this improves the previous results, as suggested by Levy and Bullinaria (2001)'s study.

The question of what basis to use is directly related to that of the number of dimensions to use. Indeed, if we choose an appropriate basis, we might not even need to reduce the semantic space's dimensionality (Levy and Bullinaria, 2001) in order to get good results. We will compare the results of not reducing our dimensionality with the method of truncating the very low frequency words from the basis (Levy et al., 1998). If these results are not satisfactory then we will test some more advanced dimensionality reduction methods such as SVD used by (Landauer and Dumais, 1997).

1.4.4 Co-occurrence count parameters

There are three main parameters that one must take into account when creating vectors of word co-occurrence. The creating of these vectors is done by counting the occurrences of context words (also called base words) neighbouring the target words. The actual counting is done through what is called a *Window*. There are different parameters of the window that one must take into account during the vector creating process (Patel et al., 1997). First of all, the *window size* has to be specified. This describes the number of words around the target word that we take into consideration. If we have a window size of 4, for example, we will consider 4 closest words to the target. This leads to the definition of a *window type* which defines how we choose the neighbouring words. There are four different types of windows:

- *Left only* - counting the context words occurring only to the left of the target.

- *Right only* - counting the context words occurring only to the right of the target.
- *Left plus Right* - counting the context words occurring of either side of the target.
- *Left and Right* - counting the context words occurring on both sides of the target word, as different elements of the vector for the word. Thus the context word occurring before the target word will be a different element in the vector of co-occurrence, than the same context word found on the other side of the target word.

It is important to note that the window type influences the dimensionality of the semantic space. Indeed the first three types of windows lead to a N dimensional space, whereas the last one implies a $2N$ dimensional semantic space, where N is the number of context words. We will test our model with both a *left plus right* and a *left and right* window type, of a constant size of 10. We will of course keep the window type which has given us the best results.

1.5 Conclusion

All in all, we have introduced semantic space models, that are used for a variety of tasks and appear in many different contexts such as search algorithms, learning theories, priming effects, etc. We have also discussed the strength and weaknesses of such models, thus rating their usefulness to our project. The specific semantic space model that we have chosen, has also been presented, and specific important design decisions have been highlighted. Finally we have concluded with a short presentation about the uses of individual differences detection and one of the main methodologies (the IAT) that is used for such tasks. The main question that remains unanswered at this stage, is if indeed individual differences are implicitly transmitted through language. If this is the case, the automatic extraction of this information may be of much use for many belief related psychological studies and Artificial Intelligence applications. All the beliefs transmitted through language constitute a major part of the culture that is gained by people. Being able to identify this cultural baggage that is implicitly passed around is very important for any program wishing to interact with humans using natural language.

Chapter 2

The program

In this chapter, we will look closely at the implementation of the Semantic Analyser program, and the Graphical User Interface (GUI) which is provided separately from it. This GUI enables a more user friendly display of the results for demonstration purposes.

2.1 The choice of the Language

The whole program, including the GUI, was written in Common Lisp using the LispWorks IDE. Common Lisp is a general purpose programming language that supports many different programming paradigms such as the functional programming and object-oriented programming. The Common Lisp Object System (CLOS) is a set of macros enabling easy object oriented programming. The CLOS extension to Common Lisp has been used heavily for building the GUI with the CAPI library provided by LispWorks. Common Lisp has been chosen as the programming language for this project for many different reasons ranging from the programming language itself to the programming environment and the software process model that it supports easily.

Firstly, as the program's main data structures are lists and hash-tables, thinking about the algorithms in Lisp, where the basic data structures are lists, was very instinctive and natural. The program runs a 'window' (which is a fixed length list) over the text, and counts word co-occurrences. After text has been analysed, similarity measures are computed via n-dimensional vector arithmetic operations, which are trivial to implement in Lisp compared with other languages. This is due to the fact that the implementation of these operations is very close to their mathematical notation. For example, an n-dimensional dot product is mathematically expressed as: $(a_0, ..., a_n) \cdot (b_0, .., b_n) = \sum_{i=0}^{n} a_i \times b_i$ The corresponding Lisp code (with no error checking) is very similar to the mathematical equation:

```
(defun dot (a b)
  (reduce #'+ (mapcar #'* a b)))
```

whereas in a C-like language it would require explicit iteration over the two vectors, while doing the required operations. This looks much less elegant (again with no error checking):

```
int dot (float[] a, float[] b) {
 float result = 0;
 for (i = 0; i < a.length; i++)
   result += a[i]*b[i];
 return result;
}
```

The high level functions such as **mapcar** and **reduce** are especially appropriate for n-dimensional vector arithmetic, and are a core part of Lisp's strength over other languages (in this specific domain). The similar function **maphash** is very useful for applying a function all elements of a hash-table. This is used, for example, when applying the log-odds-ratio, or other normalisation function to the coordinates of each target-word. Hence choosing Lisp as an implementation language has made many parts of the program much easier to think about and thus to implement.

In addition to the implementation details, Lisp's integrated environment supports very well incremental software process models such as the different ways of prototyping and extreme programming (Extreme-Programming, 1999). A mixture of evolutionary prototyping and incremental programming has been used for this project. The language supported this with its flexibility: each small section of the program could be individually compiled (or interpreted into the working memory of the programming environment) and tested. There was no need to recompile whole files when only a small change has been made. This lead to very agile programming, with very short "write/test/integrate" loops.

Another advantage that Lisp has contributed to this project is extensibility. As the program was written as a collection of very small and task specific functions, it is easily expandable, and when loaded into a Lisp environment, provides a platform for writing other text analysing tools. For example, if another application requires the window to separate word co-occurrence counts into 'before' and 'after' occurrences, this can be easily modified in the existing code. The user can use the different functions already written by the author for text analysis which are now provided as primitive functions in this programming environment. The process is made easier by the fact that Lisp provides built in garbage collection thus letting the programmer concentrate on higher level programming issues and not deal with these details. Lisp's garbage collection, in turn, leads to faster and arguably easier development.

Though the language lacks the speed and efficiency of others, such as C, it was decided that the advantages outweigh these limitations. Indeed, with no implementation effort needed, due to Lisp's interactive prompt, the user can dynamically query the global hash-tables in order to get similarity values for different pairs of words (when a hash table is loaded). In order to achieve the same result in another language, we would have to write a separate program to execute a simplified version of the Lisp natural 'read-eval-print' loop, this would

accept queries from the user and print out the similarity results. This feature has been very useful in order to test the validity of the program easily, and to compare different results informally, before writing specific subroutines to store them on disk in a formatted form.

Finally, Lisp was also chosen as the implementation language for this project due to its arguably enjoyable nature. Enthusiasm for Lisp can be found very easily in the computer science community and more specific reasons for this can be found, among other sources, at the end of Steele Jr. and Gabriel (1993). The choice of the programming language has contributed to making the programming side of the project enjoyable/fun.

2.2 The algorithm

The semantic analyser program in its most basic form can analyse a single stream of data, and then answer queries about the similarity between different words from the information gathered. We will concentrate on describing the algorithms used to analyse the text, and in a second part, we will explain the working of the different possibilities for the actual display of the similarity between words.

2.2.1 Text Analysis Algorithms

The algorithm for analysing a text, or a corpus of texts and gathering the information necessary to calculate word similarities, comes in different flavours. All the versions are very similar, but are better suited for different domains. The basic function **analyse** is well suited for a single, medium sized stream (tested on files up to 10 MB). **Analyse-files** is more appropriate for analysing a list of medium sized files. Finally, **analyse-part** is used for analysing very large corpora of data, and includes a *checkpoint* facility, in case of failure: it lets the user reload a partly completed hash-table, and resume analysing the corpora, using that table. This has been very useful for analysing the British National Corpus over an Internet connection, where disconnections from the stream are possible. The basic algorithm used for the **analyse** function is as follows:

Input:

- A text stream (opened for input)
- The size of one side of the window (default value: 10)
- A set of base-words (default value: the list taken from Lowe (2000))
- A set of target-words (default value: those taken from Balota and Lorch Jr. (1986))
- A pathname where to save the global variables

- A 'Gutenberg' boolean which specifies if there is need to strip the 'Gutenberg Project' header from the files (must be set to true only when analysing a text from the 'Gutenberg Project' database (Project-Gutenberg-Literary-Archive-Foundation, 2006).

The algorithm itself consists of the following steps:

1. Make a window of size 2N+1 (N words on each side of the word we will be analysing) where N is given by the user. We fill the window with N+1 words from the stream, the rest is kept empty.
2. Initialise the global variables:
   - *corpus-length* is initialised to 0
   - *word-count* to a hash-table containing 0 for every base word and target word that we use.
   - *target* to a hash-table containing a single hash-table for each target-word. Every one of these hash-tables is initialised with 0 in every slot containing a base-word.
3. Loop until the middle word of the window is empty (i.e. until we finish parsing the stream):
   (a) Call the function **read-one** on the window. This updates the values stored in the global hash-tables depending on the co-occurrence count.
   (b) Increment the *corpus-length* count
   (c) Move the window one word forward
4. Apply the log odds-ratio to all the elements of every target word's hash-table.
5. Save the global variables (*target*, *word-count* and *corpus-length*) in a file.

The other functions' algorithms are but simple variations of this basic one with only a few minor changes, such as not initialising the global variables but loading them from a file in order to resume computation, not using the 'Gutenberg' boolean and not applying the log odds-ratio before saving the *target* hash-table, so that the computation may restart from that point. The main loop (step 3) is identical in every case.

The algorithm for the **read-one** function is quite trivial, but we nevertheless present it formally here:

1. Extract the middle word from the window
2. If the word is a target word, get its hash-table from *target*, else return from function.

3. For every word in the window, if it is a base-word increment its associated value in the target word's hash-table.

This algorithm can be easily extended to deal with all different kinds of windows: windows which require separate counting of words before and after the target words, etc. The result of running the algorithms listed above on some text is the *target* hash-table, which we can then query (after the log odds-ratio has been applied to it) in order to get information about the similarity between different target words.

2.2.2 Similarity calculation

After the text has been analysed and the log odds-ratio calculation has been applied to the *target* table, we are left with a hash-table containing all the target words that we have analysed. Every target word is represented by a hash-table of its own, containing the co-occurrence count for each of the base-words. The similarity between two words, in our semantic space, is measured by the cosine of the angle between them. This value is proportional to the similarity between the words.

In order to calculate the cosine of the angle between two words, we first have to represent them as vectors instead of hash-tables. This is done quite trivially. We remind the reader that the cosine of the angle between two vectors is related to the dot product of these vectors by:

$$a \cdot b = |a||b|cos(\theta)$$

where θ is the angle between a and b and $|a| = \sqrt{\sum_{i=0}^{n} a_i^2}$. Hence, if $|a| = |b| = 1$, i.e. a and b are unit vectors,we have:

$$a \cdot b = cos(\theta)$$

Therefore in order to calculate the cosine between our two vectors, we first normalise our vectors:

$$\hat{a} = \frac{a}{|a|}$$

where $\hat{a}$ is the normalised form of the vector a.
The final formula for calculating the cosine, as implemented in our program, is:

$$cos(\theta) = \frac{a}{|a|} \cdot \frac{b}{|b|}$$

When the user queries the *target* hash-table, which acts as a database for the similarity between the target words analysed in the text, the hash-tables representing the two words are transformed into vector form (i.e. simple lists of numbers). The cosine between the two words is then calculated via the equation above, and is returned to the user. The *target* table can also be queried in a more user friendly way, using the Graphical User Interface (GUI).

2.3 The Graphical User Interface (GUI)

Discussion of why there is a Graphical user interface, the initial though for its use, what it ended up being (demonstration tool). Additions to be made to the GUI (integrated visualisation tool).

The GUI has been programmed using the CAPI library that is distributed with LispWorks. This is the only part of the project where object oriented features of Lisp are used, and where we make use of functions not available on all Common Lisp distributions.

Initially, the GUI has been created in order to give the program a user friendly front end, where text can be analysed or hash-tables can be loaded from files, and where the users can query the hash-tables in different ways. Subsequently, it was decided to exclude the text analysing facilities. This was done as the program was used to analyse many text files in a single corpus. The process is time consuming and more easily done through the Lisp command line listner. Instead of analysing texts, the user will have to only use the file loading capabilities of the GUI in order to query results out of already pre-analysed texts. The actual analysis will be done from the command line. Due to the fact that very large corpora of text can take a long time to analyse[1] and is rarely in a single file, it has been decided that the analysis phase should be well separated from the querying about word similarity phase.

The GUI will be most useful for demonstrating the back end capabilities of the program. It enables three types of similarity comparison:

- Comparing a single word to a set of other words (up to a maximum of 20 other words). This is called comparing words *manually*.
- Comparing two sets of words pairwise (again to a maximum of 20 words in each set) i.e. comparing together the first words of each set, then comparing the second word of each set, and so it goes.
- Comparing a set of words to itself i.e. every word in the set if compared with every other word (there is no maximum limit to how many words can be in this set, but one must remember that this operation works in exponential time with regards to the number of words in the set: there might be a significant delay to get the results if this number is high.)

The third option is the one most useful for pure research (as opposed as demonstrating the program) and is the only operation which created a file, and requires the user to load a file (containing the set of words to analyse). The output of the operation is a .csv (comma separated value) file that is openable with any spreadsheet editing program (such as Microsoft Excel or StarOffice spreadsheet). The file is created in the same folder as the loaded file, and has the same name with the suffix -sim.cvs in order to distinguish that it is the file containing the

[1] about forty five minutes for the totality of the British National Corpus, more if it is analysed across a network. This value is approximative and depends, among other variables, on the number of basewords used, the size of the window and the number of target words.

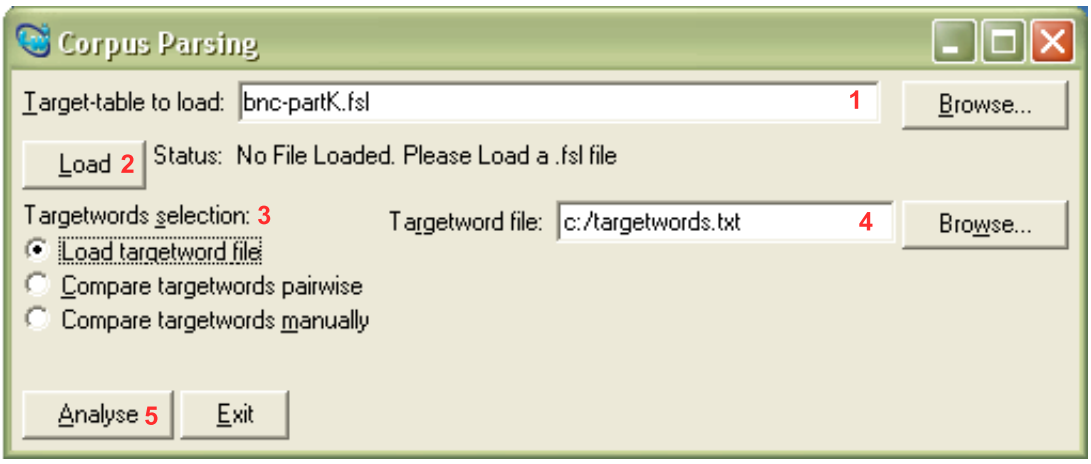


Figure 2.1: the GUI when comparing a set of words to itself

similarity values. The two other forms of comparisons require the user to type the number of words he wishes to analyse in addition to what they are. The GUI changes dynamically to provide space to type for the selected number of words and to view their similarity values.

As the GUI is for demonstration purposes, and has only a very small number of functions, there is no separate user manual to explain it. Instead we present here, in brief, its main components and capabilities. We include three annotated screen-shots, one for each possible similarity operation.

Referring to figure 2.1 we have:

1. The target table to load into the Lisp environment. This is a .fsl file that has been previously saved by the **analyse** function or one of its variations. The user can either Browse his system to find such a file, or directly write its address.

2. The Load button loads the selected target table into the system. The status display next to it informs the user of the state of the system: if a target table is loaded, and the size of the window when the text was analysed.

3. The three similarity calculation options. When these change, the GUI dynamically modifies itself to suit the task the user wants to perform.

4. The Target-word file selector lets the user select a file containing all the target-words that he wishes to compare pairwise. The target-words in the file must be separated by spaces.

5. The Analyse and Exit buttons are responsible for the actual behaviour of the GUI. The Exit function closes the window, whereas the Analyse button calculates the similarity, depending on the options selected above.

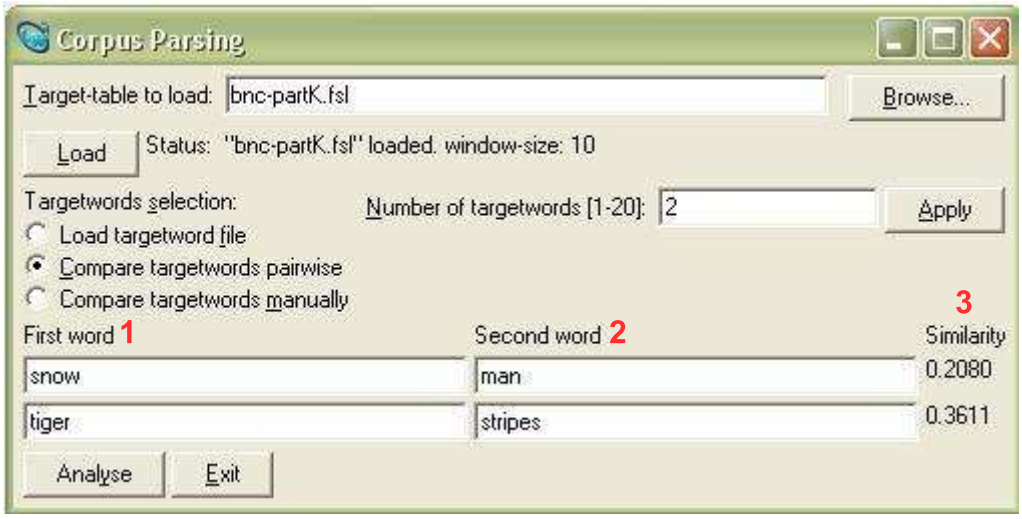


Figure 2.2: The GUI when comparing words pairwise

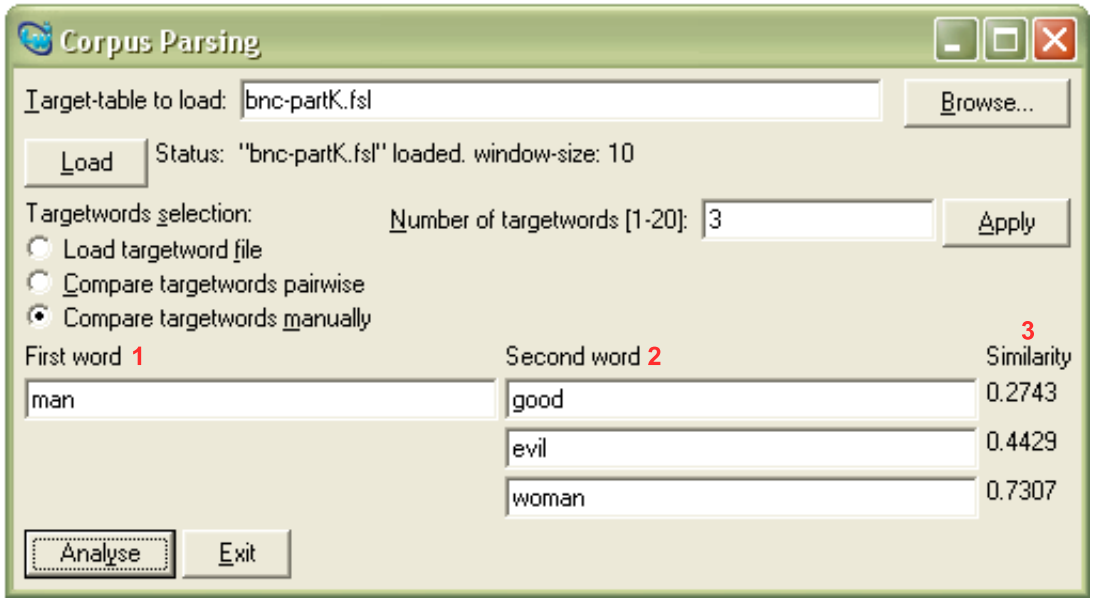


Figure 2.3: The GUI when comparing a single word with many others

Referring to figures 2.2 and 2.3, we have:

1. The first word (or set of words) to be compared. These must be typed by the user in the available text boxes (after the number of words to compare has been chosen by the user, by typing that number in the available text box and pressing the Apply button).

2. The set of words to compare the word (or words) in 'First word' column. If there is only one word in that column, every word in 'Second word' column is compared with that word. In the other case, the words are compared pairwise.

3. The similarity display, which holds the values of the similarity between the compared words only once the Analyse button is pressed.

2.4 Program's Design decisions

During the course of the project the exact implementation of the program has evolved greatly, with the iterations over the software process model. Most of the changes were low level and initially depended on optimising the speed of execution of the analysing part of the program. The main design decisions that will be discussed here are: the use of lists versus the use of vectors, the use of function closures and the move from a two hash-table representation of a target word to a single window representation and the different versions of the analysing function. We will also mention the tools available in Lisp which have helped us make informed decisions about the status of the program and the resources it was consuming. Finally, we will present the different variations of the basic **analyse** function that we have provided, with a small justification for each of them.

As the basic algorithm to analyse text depends on a fixed size window through which the stream is analysed, it has been hypothesised that using a fixed size *queue* data-structure instead of the basic list would be much more efficient with regards to speed and space. A queue data structure differs from a list in that the elements inside it are hidden, except the *head* and *tail*. Programming-wise, our conjuncture was that by not having to resize the queue as we had to with the list (which by definition vary in size) upon each call to the **mv-one** function, the program as a whole would run faster and require less allocation and garbage collection. This hypothesis has been tested using the efficient queue implementation proposed by Waters (1991). Unfortunately, as in our program we often have to access all the words inside the queue in order to count their co-occurrence, the advantage of using this more complex data-structure was outweighed by the high cost of accessing all the elements of the queue. We thus reverted to the initial, and more simple list implementation of our program, and moved on to try and optimise another part of the program.

The second major optimisation that was attempted regarded input/output issues and concentrated on the **getword** function which returned the next word

in the stream. The initial, simple, algorithm for this function was to read from the stream a character at a time, until a whitespace was reached. The function then returned the string that it has read, deleting any punctuation marks and making the whole string lowercase. Strings made only of punctuation marks or numbers were disregarded. With the knowledge that in many cases bulk reads are more efficient than reading one character at a time from a stream, the algorithm was rewritten. The new version of the algorithm made use of Lisp's powerful feature of explicit function closures. A function closure can be used to store a global variable that only some functions can access. In our algorithm, we used a single variable which is accessible only by **getword**. This variable is used to store the words read from a whole line of the stream, thus resulting in a bulk read from the stream. When **getword** is called, if the variable is empty, than another bulk read must be performed, otherwise a word is taken out of the list and returned. In Lisp, this function is coded as:

```
(let ((words nil))
  (defun getword (stream)
    (cond ((null words) (setf words (getword-aux stream))
           (if (null  words) nil (pop words)))
          (t (pop words)))))
```

Where **getword-aux** deals with reading a whole line from the input stream, dividing it into words, and deleting punctuation marks and HTML directives. The variable *words* is only accessible by **getword** and stores the list returned from **getword-aux**. Thus the stream is accessed less often, and for bigger reads. This has lead to a great increase of speed in the execution of the program. After concentrating our optimisation efforts on the low-level side of the program, we have transferred our attention onto the more high-level components such as the representation of the target-words' co-occurrence counts.

The initial version of the program made a distinction between words counted before and after a target word. This lead to each target word being represented by two distinct hash-tables, one to count the 'before' co-occurrence (i.e. the words that occurre before the target word in the window) and the 'after' co-occurrences. The final dimensionality of the representation of the target word in our semantic space, was therefore 2N where N is the number of base words that we have used. As we will discuss in chapter 3, this added more noise to our results which in turn lead to a change of approach. The distinction between 'before' and 'after' co-occurrence counts was then erased, and each target word was represented as a single N dimensional vector of total co-occurrence counts ('before' counts + 'after' counts). From a programming point of view, this has lead to a faster and more memory efficient implementation. All these optimisations would have been much harder to measure if it was not for the tools that Lisp and LispWorks provide for this purpose.

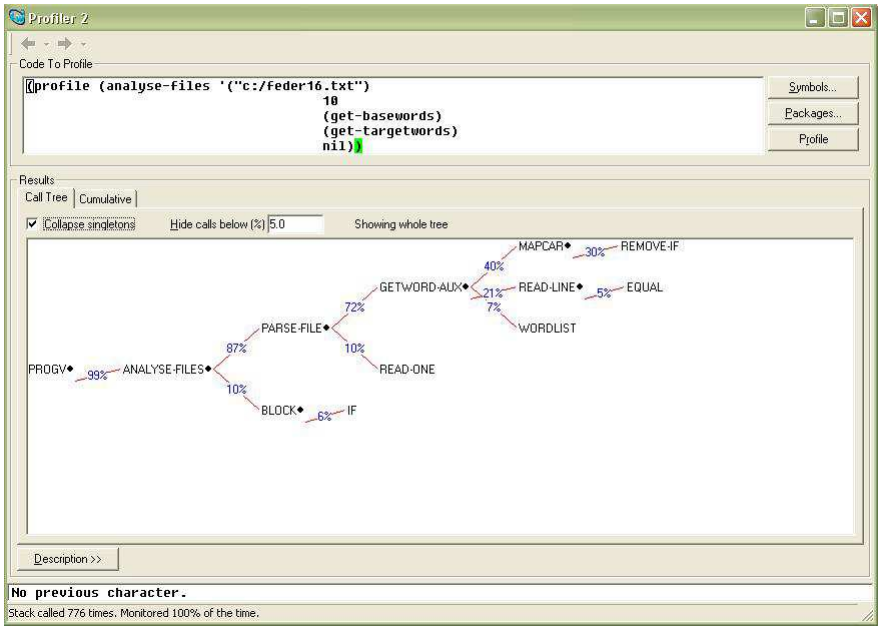
There were two tools that were used in order to optimise our program: the **time** function available on most Lisp implementations, and the profiler that is specific to LispWorks distribution. Lisp's **time** function gives statistical information about the runtime of a function including elapsed real time, and

storage management statistics (Steele Jr., 1990). An example output of the time function in the LispWorks environment is given below.

```
Timing the evaluation of (ANALYSE-FILES (QUOTE ("c:/feder16.txt"))
10 (GET-BASEWORDS) (GET-TARGETWORDS) NIL)

user time    =      5.397
system time  =      0.030
Elapsed time =   0:00:06
Allocation   = 24713768 bytes standard / 14273281 bytes conses
0 Page faults
NIL
```

The main statistics that we have compared between different versions of our programs is the Elapsed time. The profiler is used to provide more detailed statistics about the functions that are called. It is used to know what functions are taking up most of the total execution time of the program. The information the profiler gives is: the number of times a function was called, the number of times the function was found in the the stack (both in absolute terms and as a percentage of the total scans of the stack) and the number of times the function was found on top of the stack (again in absolute terms and as a percentage)(see LispWorks User Guide). A sample output from the profiler looks like:



This display of the profiler only shows the very basic information: i.e the percentage of the program running time that was spent on each function. The profiler thus helped us find which functions to concentrate our optimisation on,

whereas the **time** function was used to evaluate how effective the optimisation was, and to compare different versions of our functions.

The main functionality of the program, that of analysing text, was implemented with several small variations. We will list here these different versions of the basic program and a short description of their main features and uses.

- **analyse** is the basic function which analyses a medium size stream (usually opened from a single file). Functionality includes saving the global variables to file, after having applied the log odds-ratio.

- **analyse-files** a slightly more advanced function enabling the analysis of a list of files. The global variables are again written to a file after the log odds-ratio has been calculated. The actual analysis of each individual file is done via the **parse-file** function.

- **analyse-part** is the most flexible function. It analyses a file given file list, but does not finalise the global hash-table. It saves the global variables in a file, which can subsequently be loaded in order to resume the analysis. Again the analysis of the individual files is done via the **parse-file** function. Before querying the *target* hash table in order to get similarity values for pairs of words, the log odds-ratio must be applied to *target*. This is done by the **map-log-odds-ratio** function.

These functions are very flexible and can be easily adapted in order to analyse texts differently: for example differentiating between a 'before' and 'after' window, or not separating singular words from plural words so that the word 'dogs' will be equivalent to 'dog'.

Chapter 3

The Validation

Once our program was developed, we had to make sure that the results it gave us were psychologically correct with respect to the priming and mediated priming results that have already been identified by other semantic space models (Lowe and McDonald, 2000). We thus replicated the Balota and Lorch Jr. (1986) and Ratcliff and McKoon (1992) priming and mediated priming results thus showing that our program and semantic space produce psychologically correct information.

3.1 Priming and Mediated Priming Studies

Semantic space models, such as the one used as the basis of our program, have been shown to be able to replicate priming and mediated priming results (Lowe and McDonald, 2000). As this of course depends on the different components of the semantic space, such as the basewords, the type of the window, etc. we needed to make sure that our model can also simulate these results. At this point it is useful to note that in traditional priming and mediated priming experiments, the reaction time (RT) of the subjects is inversely proportional to the degree of facilitation of the target with regards to the stimulus word. The lowest RT occurs in tasks involving the target and its prime, a higher RT for the target and its mediated prime and finally the highest RT for the target word and unrelated words. In our experiments, we measured the similarity between words using the cosine of the angle between them (in our semantic space). This cosine value is proportional to the similarity between the words: the higher the value, the more similar the words are. Therefore our similarity values should be inversely proportional to the RT that the experiments calculate. Attempting to replicate the priming experiments has also lead us to the final choice for the parameters of our semantic space. Experiment 1 simulates the priming results found by Ratcliff and McKoon (1992), whereas experiment 2 concentrates on the Balota and Lorch Jr. (1986) results. After presenting these two experiments, with the final semantic space chosen, we present a couple of the alternative

semantic spaces that we have tried, and compare the results obtained with each of them.

3.1.1 Experiment 1: Priming

The stimuli used in Ratcliff and McKoon (1992, Experiment 3) were divided into three groups. Each target word had a free association prime, a high t prime and a low t prime. Free association primes, were 'well known' primes to the target words (i.e. a prime taken from association norms). A t statistic for a prime is a value estimating if two words co-occur significantly more often in a text than in a random distribution of words (Church and Hanks, 1989). Thus the high t primes are words whose t statistic with the target word is high: i.e. they occur significantly more often with the prime (respectively low t primes have a lower t statistic which is still higher than unrelated words). Target words were associated with unrelated words, so as to be able to measure the priming effects of actual primes. Unrelated words were randomly chosen amongst low t primes of other targets. The experimental setup was equivalent to all the other similar priming experiments. The subjects were instructed to react to the words given as a stimuli on a computer screen. RT to each word was measured and it was noted what the preceding word was (i.e. a prime of a certain type or an unrelated word). The experiment's results showed that the fastest RTs occurred with the free-association primes, followed by the high t primes. Low t primes seemed not to give a significant decrease in RT, but still more than the unrelated words.

Experimental setup

In order to replicate this experiment, we ran our program on the contents of the British National Corpus (BNC) as available to the University of Bath. The BNC includes about 100 million written words, extracted from various sources. We have used it as a representative sample of (English) language use. Our semantic space was constructed by analysing the BNC using the following components:

- Base words: the set of base words identified by Lowe and McDonald (2000).
- Target words: the target words were the stimuli used in Ratcliff and McKoon (1992, Experiment 3).
- Window size: 10 words on each side of the stimulus items. The co-occurrence counts were then added together (i.e. not differentiating 'before' and 'after' occurrences).
- Normalisation function: we used the positive log odds-ratio in order to normalise the word vectors that we had after the text analysis.

Once the semantic space was constructed, we calculated similarity values (i.e. the cosine) between target words and their different primes. We also calculated the average similarity between the target word and 10 unrelated words.

	Free association primes	High t primes	Low t primes	Unrelated
R&M (RT in ms)	500	528	532	549
space (cosine)	0.657	0.571	0.527	0.426

Table 3.1: Comparison between Ratcliff and McKoon's (1992) experiment 3 and our program's replication of the experiment

Results and Discussion

As the t statistics of words is in fact a measure of their co-occurrence, we hypothesised that our program will be able to show higher similarity values between target words and high t primes, slightly lower values for low t primes and very low values for unrelated words. As shown in Table 3.1 our results correspond to the findings of Ratcliff and McKoon (1992, Experiment 3) and agree with our hypothesis. In addition the analysis of variance conducted on these results has shown that the differences in similarity are indeed reliable and not due to random distribution of values ($F_{3,156} = 30.56, p < 0.001$).

3.1.2 Experiment 2: Mediated Priming

Balota and Lorch Jr. (1986, Experiment 1) clearly displayed mediated priming effects, using the standard experimental setup. The stimuli that was used consisted of word triads: a target word, a direct prime and a mediated prime. In order to identify the facilitation words unrelated to the targets were used. For a target word, an unrelated word is simply a prime of another word. The experiment showed that mediated primes indeed facilitated their target words more than unrelated words, but less than direct primes. This was done again by timing subject's RT to the different words. When a word was preceded by it's prime, or mediate prime, the RT was shorter than when it was preceded by an unrelated word.

Experimental setup

Our experimental setup was much the same as in Experiment 1. The target words of course were changed to the stimuli used by Balota and Lorch Jr. (1986, Experiment 3). When the analysis was done, we calculated the similarity values between each target word and their direct primes, mediated primes and unrelated words. We randomly selected 10 unrelated words and averaged the similarities between each of them and the target word, in order to get a similarity value for our word and unrelated words.

Results and discussion

As expected, our results mirrored Balota and Lorch Jr. (1986, Experiment 1). These are summarised in table 3.1.2. The priming results were clear: related

	Related	Mediated	Unrelated
B&L (RT in ms)	527	567	574
space (cosine)	0.525	0.485	0.455

Table 3.2: Results from Balota&Lorch experiment and the replication of that experiment using our program.

Basis	Related	Mediated	Unrelated
normal	0.525	0.485	0.455
augmented	0.673	0.647	0.630

Table 3.3: Comparison between priming results obtained using Lowe's basis and the same basis augmented with the 147 most frequent words.

words have a greater similarity value than mediated primes which in turn have greater similarities to the target word than unrelated words. This clearly follows our predicted results and an analysis of variance (ANOVA) conducted on these values showed that the differences seen between the different priming phenomenons, within our model, were indeed significant ($F_{2,143} = 6.87, p \leq 0.001$). Now that we had confidence in the fact that our model can indeed detect priming effects, we could move on to try and optimise the different parameters of the semantic space model that we will use to identify individual beliefs.

3.1.3 Choice of parameters for the Semantic Space Model

In order to choose optimal values for our semantic space model, we have conducted a few tests with the Balota and Lorch Jr. (1986, Experiment 1) stimuli. We have kept the values which gave us the best results. This lead us to a very similar semantic space model as Lowe and McDonald (2000).

We have started by fixing our window size to 10, and varying the context words used. We tried both the base words used by (Lowe and McDonald, 2000), and then augmented them with the 147 most frequent words as suggested by Levy and Bullinaria (2001). The results (see Table 3.1.3) show that the first choice for the base yielded the best priming results (the difference between related primes and unrelated words is the greatest). In both cases, the priming results were reliable ($F_{2,143} = 6.87, p < 0.001$ and $F_{2,707} = 11.84, p < 0.001$).

Subsequently, we kept the base words fixed and varied the window size between 6 and 12. The best priming results were observed when the window size was 8 (see table 3.4). This led to the most relevant difference between primes and mediated primes, as well as between mediated primes and unrelated words. The priming conditions were reliably different ($F_{2,143} = 10.07, p < 0.001$) under these conditions. Subsequently, when exploring individual differences (chapter 4) it became apparent that results collected with a window size of 10 were much more intuitively correct for that task. We have thus kept a window size of 10 despite the slightly less optimal priming results. We have not tried varying both the base words and the window size, which would have maybe led us to a

Window size	Related	Mediated	Unrelated
6	0.312	0.255	0.224
8	0.335	0.277	0.235
9	0.341	0.282	0.239
10	0.525	0.485	0.455
11	0.537	0.498	0.466
12	0.359	0.298	0.255

Table 3.4: Result of comparing the effect of different window sizes on the priming and mediated priming results.

Normalisation function	Related	Mediated	Unrelated
none	0.483	0.448	0.365
frequency	0.395	0.349	0.268
log odds-ratio	0.525	0.485	0.455

Table 3.5: Result of comparing the effect of different normalisation functions on the priming and mediated priming results.

different choice of a semantic space. As we have seen in Experiment 1 and 2, our choice semantic space gave us quite good results in priming tasks.

The last parameter that we varied was the normalisation function. This is applied to the raw co-occurrence counts after the text was analysed and before the computation of the similarity between words. The other parameters were fixed (window size: 10 and the context words agreed upon above). The first normalisation function that was tested was the identity function, this means that we calculated similarity values between words using the raw co-occurrence counts. Subsequently, We have normalised the raw counts by dividing them by the the overall frequency of occurrence of the target word (Levy and Bullinaria, 2001). We compared these techniques to the log odds-ratio Lowe and McDonald (2000) method and concluded that the latter yielded the best results.

Without having compared every possible combination of the different context words, window size and normalisation function, we can still say that the factors which we chose gave us relatively positive results. These factors are: the log odds-ratio normalisation function, with a window size of 10 and the 536 context words identified by Lowe (2000) as a basis.

Chapter 4

Exploration

Now that we are convinced that our program can detect the priming phenomenon, and therefore word facilitation, we can use it to try and detect individual differences in beliefs. We will start by trying to find the basic results that have been put forward by the IAT before trying our program on unanalysed data such as the Bible and the works of William Shakespeare.

4.1 Retracing the steps of the IAT

The Implicit Association Test (IAT), as discussed in section 1.2.4, measures individual differences in beliefs. The first studies conducted with this test have shown some interesting associations and biases (Greenwald et al., 1998). The IAT was sensitive to "near-universal evaluative differences" in addition to "consciously disavowed evaluative differences" (Greenwald et al., 1998). In other words, the IAT has managed to detect individual differences, both in terms of expected results (such as the fact that most people associate flowers with pleasantness and insects with unpleasantness) and less expected results (such as the fact that self-proclaimed unprejudiced White subjects still have a racial bias towards White people/against Black people). Using our program, with the semantic space discussed in the previous chapter, we have replicated to a certain extent these findings of the IAT.

Experimental Setup

The British National Corpus (BNC) was analysed with our program. The BNC is considered as a representative sample of how language was used today. Thus the data gathered from this analysis is assumed to be representative of the beliefs commonly held in today's British culture. The target words used included: a set of flower names, a set of insects names, a set of weapons, a set of musical instruments, a set of pleasant words and a set of words deemed unpleasant. All these were taken from the stimuli list of Greenwald et al. (1998). In order to have

Category	Pleasantness	Unpleasantness
Flowers	0.147	0.141
Instruments	0.162	0.140
Insects	0.138	0.154
Weapons	0.165	0.206

Table 4.1: The average similarity between categories of words and pleasant/unpleasant words.

a more intuitive set of stimuli, we have augmented each set with the singular and the plural of the set's name (e.g. the set of flower names was augmented with the words 'flower' and 'flowers'). Due to the ambiguity of the word 'instrument' we have not added it to the target words set. The words 'white' and 'black' were also added in order to test for any racial bias. Other than this change in the target words, the program's semantic space was equivalent to the one identified in the previous chapter.

Results

In order to gather results, after analysing the BNC with the specified target words, we averaged the result of comparing the similarities between each of the categories and the pleasant and unpleasant words, in order to rate the relative pleasantness of the category. Words which occurred less than 100 times in the BNC were discarded, as they reduced the relevancy of the data gathered (for the insects category for example $F_{1,52} = 1.01, p = 0.319$ when keeping the low frequency words and $F_{1,28} = 8.25, p = 0.008$ when discarding them). The results are summarised in table 4.1. As expected, and shown by Greenwald et al. (1998), the flowers and instruments categories came out to be more similar to pleasant words than unpleasant words. The results were all reliable, apart from the comparisons done with the flowers category ($F_{1,22} = 1.16, p = 0.293$). Insects and weapons came out to be more similar to the unpleasant words. Finally, the racial bias that we were looking for was identified, but only slightly. The word 'black' was more associated to unpleasant words (by 0.0194[1]), the word 'white' was also slightly more associated to unpleasant words than to pleasant ones but the bias was smaller (0.0101) than the one for 'black' thus showing that 'black' is indeed more associated to unpleasantness but not by much ($0.0194 - 0.0101 = 0.093$).

Discussion

In order to be able to detect a racial bias more strongly, the stimuli of the experiment should be changed. The IAT experiments used typical black and

[1]This was calculated as $|similarity(black, pleasant) - similarity(black, unpleasant)|$ where the similarity function takes a word and a set of words and returns the average similarity between it's first argument and all the elements of the second argument.

white American first names (Greenwald et al., 1998) and later simply used pictures of black or white faces (Mitchell et al., 2003). In order to get better results, it would be possible to try and use these first names as target words. The frequency of the names in the BNC may present a problem for the final analysis, and instead of averaging the pleasantness/unpleasantness results for each name, summing the raw co-occurrence counts and subsequently treating the resulting word as if it were a single word (containing all black or white American names) when calculating similarity values might lead to better results. The results regarding 'universal evaluative differences' such as the relative unpleasantness of insects and weapons and the pleasantness of instruments have however shown that the program can indeed be used to analyse differences in beliefs with some degree of confidence.

4.2 Analysing evolution of beliefs: from the Bible to Today

For the purpose of demonstrating more original results using the program, rather than replicating different experiments, a number of words have been analysed in texts which differ greatly in the time in which they were written. These texts were the Bible (Douay-Rheims Version), William Shakespeare's first folio (the first 35 plays) and the BNC. The relative similarity between the words in each corpus was calculated, in order to observe if any radical shifts in beliefs could be identified. The hypothesis was that the evolution of the language use between the time each of the texts was written should be observable via the evolution of the different similarity ratios between the words. One must note, that the Bible and especially the new testament, has an important influence on individual beliefs of people in western cultures. This influence was even stronger in Shakespeare's time (the 16^{th} century) as Christianity was very influential in Britain and accepted no criticisms. It was thus suspected that a gradual loosening between the beliefs present in the bible and beliefs present in the Shakespeare corpus would be observed. The BNC data was deemed to show further loosening of these beliefs as some of the ideas in the bible have started to be thought of differently (such as the role of women which has changed greatly in the beginning of the 20th century).

Experimental Setup

The semantic space used for the analysis was similar to the one used in experiment 4.1. The only difference was the stimuli and texts that were used in the analysis. The target words used were: black, white, good, bad, right, left, life, death, man, woman, men, women, dog, God, gods, dogs, war, peace, evil. The analysis was run first on the Bible, then on the the first Shakespeare folio and finally on the BNC. For each text, a force directed graph was outputted, showing graphically the similarity between words (words closer together were more similar). Words which occurred with very low frequency in the texts (less

than 100 times) were ignored. Target tables were also saved onto files so as to enable subsequent similarity calculations if needed.

Results and discussion

When looking at the evolution of women in the texts one can notice a relative change. In the Bible, the word 'woman' is closely associated with the positive words 'good' and 'life'. In the Shakespeare corpus, the singular 'woman' becomes suddenly more associated with the word bad than with the other words. The plural 'women' is still associated with positive words such as 'peace', 'good', and 'god'[2]. Finally, in the BNC, both the singular and the plural of 'women' are closely associated with the words 'man', 'men' and 'life'. Relatively, it still seems that the word 'woman' is still closer to the negative words 'bad' and 'evil' than to the positive words 'good' and 'peace'. One should not conclude from this that women are seen negatively as this is also true for the words 'men' and 'man'. It is possible to see the shift from 'woman' being highly associated with positive words, to a more balanced position closer to the word 'man' as the acquisition of a more equal status for women: the words 'woman' and 'women' are being used in similar contexts as the words 'man' and 'men'.

It is also interesting to notice the fact that even though culturally, it seems though the word 'white' should be close to positive words such as 'good' it is not the case. Even the word 'black' is not consistently similar to negative words (indeed, it is closer to the word 'life' than to the word 'death' for example). A more detailed inspection of the similarity values leads to better results. In order to decide if a word is positively or negatively connotated it should be sufficient to say that on average it is closer to pleasant or very positively connotated words. This has been done using the list of pleasant and unpleasant words from Greenwald et al. (1998), and the results showed that the word 'black' was closer to unpleasant words than to pleasant words (0.17 and 0.15 similarity average). The same test on the word 'white' showed a very small bias towards unpleasant words as well, but the difference was smaller (0.14 and 0.13). This seemed to agree with the simple results of figure 4.3.

Another observation that can be made on the analysis is that in the Bible, dogs were not considered to be moral agents. Indeed, the word 'dogs' is very weakly associated with the other words displayed, and is quite far from moral words such as 'good', 'bad' and 'evil'. In addition the association between 'dogs' and other moral agents such as 'man', 'woman', 'men' and 'women' is also weak. In Shakespeare's texts the situation have evolved somewhat, with the word 'dog' closer to 'women', 'man' and 'good'. The BNC results in figure 4.3 clearly show that both 'dog' and 'dogs' are very much closer to the other moral agents as well as to the word 'evil' which clearly demonstrates that they are regarded as moral agents.

[2] As the text has a clear Christian ideology, it is assumed that the word 'God' has a positive connotation. This is supported by the analysis in table 4.2 which shows that this word is very closely associated with the word 'good'

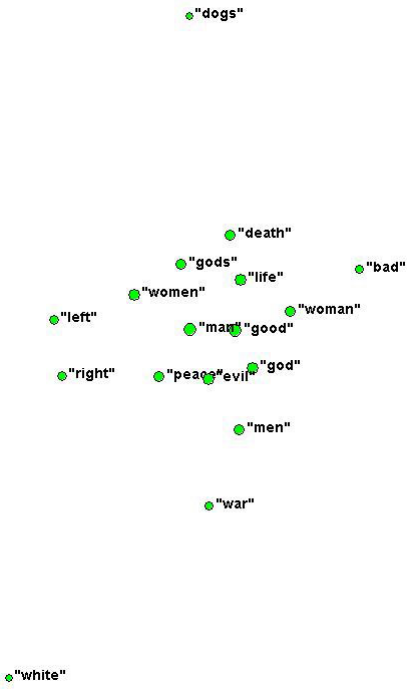


Figure 4.1: Force directed graph plotting some of the words analysed in the Bible and their similarities. Similar words are closer together than less similar words.

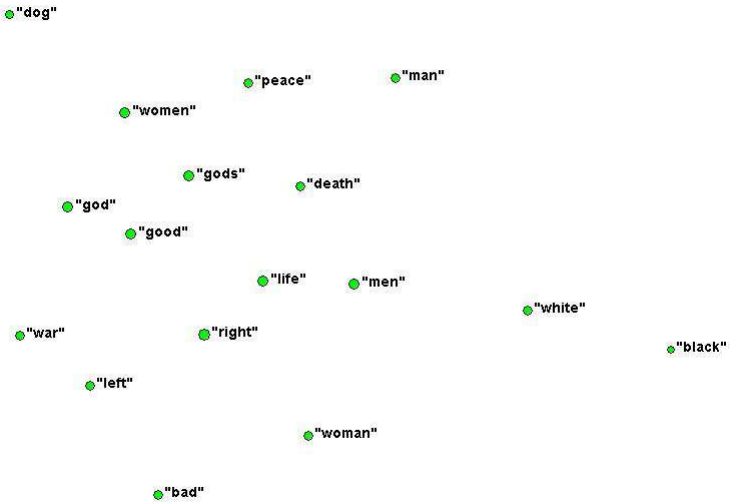


Figure 4.2: Force directed graph plotting some of the words analysed in the first 35 plays of William Shakespeare.

Figure 4.3: Force directed graph plotting some of the words analysed in the British National Corpus.

A final interesting fact that can be deduced from the results presented above is that men and women were not described by colours such as black and white in the Bible, but this again changed over time. Indeed, the word 'white' is very weakly related to 'man', 'woman' and their plural in figure 4.1. The word 'black' does not even occur often enough in the Bible to be taken into account in the analysis. On the other hand in the Shakespeare folio, 'white' is more closely associated to 'men' and 'woman' but is still relatively far from 'man' and 'women'. Finally in the BNC analysis, 'black' and 'white' are both much closer to 'man', 'woman' and their plurals. This could be taken as a sign that people tend to categorise each other more and more by skin colour. This observation could be extended by adding the words 'brown' and 'dark' to the experiments stimuli, as it is common (in the 21^{st} century) to say that someone has dark or brown skin.

All in all, the program seems to perform adequately. From the results above, it can be seen that some optimisation of the semantic space is probably needed in order to perform even better in these operations. The unintuitive results about the colour white and the racial information that was extracted from the BNC corpus supports this claim for further optimisation needed. Never the less, the results are still quite encouraging, showing that it is possible to automatically explore individual differences through language use.

Chapter 5

Future Work

As stated in the previous chapters, the program itself and research into the automatic detection of individual beliefs through language use is still not perfect and there is much possible work to do. We will now present possible extensions to the program presented in chapter 2, before presenting possible work to be done in the more global field of work of the detection of individual differences through language use.

5.1 Extensions to the program

The possible extensions that can be made to the program, range from simple speed/memory optimisation (i.e. make the program consume less memory and reduce its running time on text analysis) to change the GUI in order to add an integrated visualisation tool.

In order to optimise the program with regards to speed and memory use, one must have good knowledge of LISP and of the optimisations that it performs. Rewriting some functions as tail recursive functions can lead to better results depending if the LISP implementation supports tail recursion optimisation. A better understanding of the use of *declarations* in LISP can also lead to faster code. Declarations allow programmers to pass meta-information to the compiler, i.e. information about a part of the program such as return values for specific functions or the type of a variable[1] (Lamkins, 2004). The reduction of garbage collection can also be used in order to make the program run faster. Destructive operations should be used when possible, and attention should be focused on the functions which are used most often (or take the longest to return). The

[1]The reader is reminded here that LISP is an untyped language which lets the programmer express himself with relative ease, without thinking about very low level issues when possible. Declarations break this simple LISP style by allowing programmers to reveal their intent to the compiler which can thus generate better code. The cost of doing this is flexibility, as the programmer must stick to the declarations that he has made to the compiler: if a variable has been declared an integer, it must be used as such at all times.

functions on which to concentrate further optimisations can be easily identified by the use of the profiler (see section 2.4).

The GUI can also be improved. User tests can be conducted to evaluate the ease of use of the interface and changes can be made to the position of the different elements on the screen. New capabilities, such as an integrated visualisation tool, can be added to it. The user will probably want to output force directed graphs in addition to simply calculating similarity values between words. Other output formats can also be thought of, in order to meet individual researcher's needs.

Finally, one might think about porting the program into a more cross platform programming language such as Java, or a more efficient one such as C in depending of specific needs. All in all, the program as it is at the end of this project, performs the operations required of it quite well and is useful. As with most software, additional work can of course improve it.

5.2 Further work in the detection of individual differences through language use

Optimising the program to make it more efficient, is only one side of the coin. One must also think about making it more reliable in detecting correctly individual differences. Further studies into the optimal semantic space parameters should be conducted, and the exact combination of parameters (context words, normalisation function, similarity calculation method) should be found, such that the results of the IAT and the priming experiments replication will be optimal. Once the parameters are found, other explorations into individual differences could be found. The different opinions which are transmitted subconsciously by newspapers and their effects on people who read them could be analysed. Another possible research route which is possible is the analysis of beliefs of different ethnic minorities in a country and their relation to the traditional cultural beliefs which are held in their original country, and in the country in which they live. The evolution of beliefs can also be looked at, by taking texts written in different times, and interesting conclusions can be drawn from that. Many other applications of our program can be found easily, such as following the research conducted with the IAT.

5.3 Other applications

The tool that has been implemented in this project can be used for other purposes than just detecting individual differences. A possible application is to augment natural language processing programs, giving them dynamically information about connotations of words which will make them able to respond more naturally and better 'understand' users. By calculating the similarity between words, our program can also, to some extent, be used in search facilities. Related search results can thus be found, simply using the analysis of the BNC

and finding very related words to the search word and displaying results found for them. The reliability of this task can be compared with existing facilities of search engines. These are only a few of the possible applications of the technology developed in this project, which can be very useful in the field of Artificial Intelligence.

Chapter 6

Summary

This research project has lead to the development of a program capable of analysing individual differences in beliefs through language use, with some degree of confidence. The process of building, validating and using this program has been documented in this thesis.

The underlying psychological theory was initially presented as well as an overview of different knowledge and memory theories. Spreading activation theory models the brain as an undirected graph whose vertices represent concepts in long-term memory and edges represent associative pathways between these concepts. When a concept in the graph is activated (by presentation of a stimulus), activation is spread to neighbouring vertices. This explains the concepts of priming and mediated priming which are observed as a facilitation of recognition of certain words when presented in a related context. Compound cue theory models knowledge as an interaction between short-term memory (presented stimuli) and long-term memory. If the presented set of stimuli stored in short-term memory has a high familiarity value, which depends on information contained in long-term memory, than it is facilitated. Thus both models present an explanation for the priming phenomenon. The priming phenomenon has motivated researchers into using observed facilitation in word classification or recognition in order to deduce implicit beliefs. The Implicit Association Test (IAT) was designed exactly to do this. Its success in detecting these individual differences in beliefs by using classification tasks, has inspired us to try and replicate this without explicitly testing subjects but by analysing language.

Semantic space models have a formal theoretical basis, and are used to analyse different properties of words or documents. We have used their power, which rely on counting co-occurrences of words and plotting words which are to be analysed as vectors in a high dimensional space, in order to replicate priming and mediated priming results. We then moved to using these models in order to identify individual differences in beliefs and latent cultural beliefs which are communicated implicitely by language use. The studies observing individual beliefs performed using the IAT have been replicated to make sure that we can indeed identify individual differences in beliefs. A computer program construct-

ing a semantic space was developed in order to automatically identify these beliefs. The different parameters of the semantic space model were of course optimised in order to give the most psychologically correct results, both on the priming identification tasks and the individual differences identification ones.

The program was written in Lisp and provides both a command line interface to analyse texts and a GUI that can be used to query results of pre-analysed texts. The program also lets the user query the results of analysed texts via the command line. A simple function which gives a similarity value for any two words which were analysed in the same text is provided. This similarity value is only relevant when taken in relation to other similarity values: a similarity value for a pair of words taken on its own is useless. As it is written in Lisp, the program automatically provides an interactive platform for research into text analysis using different semantic space models. It can be relatively easily extended to cater for any arbitrary semantic space that the user wishes to use. The usefulness of the program was demonstrated by analysing the evolution of cultural beliefs found in the Bible, the works of William Shakespeare and the British National Corpus (BNC) which represents the way language is used today. This exploration of belief evolution has given some interesting results, such as the fact that women become more equal to men. Following the IAT's research, it was also possible to show some racial bias against black people in the BNC, and thus in today's (English) society.

Finally we concluded the project with propositions for further improvements both on the program itself, including its Graphical User Interface, and on the applications of our semantic space model method of identifying cultural and to some extent individual differences in beliefs. These applications included improving natural language processing systems by making them aware of the connotations of different words. All in all, the project has been successful in producing an adequate text analysis tool, which nevertheless needs some more refinement in order to perform to its best possible potential.

Appendix A

Example of program interaction

This appendix is a short tutorial on how to interact with the different components of the program from the LISP listener. We will not discuss interactions with the GUI as this was described in chapter 2 section 2.3. In the following, all user typed commands will be preceded by "CL-USER X >" where X is a number representing the number of the user interaction. All the interactions that follow assume that all the defined functions were loaded into the current working memory of the Lisp session ran. In order to do this, it is sufficient to load the 'semal.lisp' file with the *load* function. Another useful note is that Lisp is not case sensitive, and commands can therefore be written in upper case, lower case or any combination of cases.

A.1 Analysing texts and saving the results

The different text analysing functions can appear quite overwhelming for the first time user, but each of them has a specific use as discussed in section 2.4 of chapter 2. A user with a little knowledge of the Lisp programming language can easily create a new function with the exact functionality required.

The simple text analysing function is the **analyse** function. It analyses a single stream and populates the ***word-count*** and ***target*** global hash tables which are used for the similarity calculation. This function also saves all the target table in a file ready to be loaded at a future date. The only minor difficulty that this function presents is the fact that the user must specify a stream as an input and not a file. A simpler version which accepts a file as an input, or even either a file or a stream, can be easily written using this function:

```
(defun easy-analyse (file
                     &optional (winsize 10)
                     (basewords (get-basewords))
```

```
                (targetwords (get-targetwords))
                (pathname "foo.txt")
                (gutenberg nil))
 (with-open-file (in file :direction :input)
    (analyse in winsize basewords targetwords pathname gutenberg)))
```

The **with-open-file** command, makes a stream from a file and binds the stream to the name specified (in this example 'in') in the body. The simple analysis of a file called 'c:/ws.txt', using the default parameters and the **analyse** function for example is achieved by this interaction:

```
CL-USER 1 > (with-open-file (in "c:/ws.txt" :direction :input)
              (analyse in))
;;; Compiling file foo.txt ...
;;; Safety = 3, Speed = 1, Space = 1, Float = 1, Interruptible = 0
;;; Compilation speed = 1, Debug = 2, Fixnum safety = 3
;;; Source level debugging is on
;;; Source file recording is  on
;;; Cross referencing is on
; (TOP-LEVEL-FORM 1)
; (TOP-LEVEL-FORM 2)
; (TOP-LEVEL-FORM 3)

;; ** Automatic Clean Down
#P"C:/Documents and Settings/Avri/foo.fsl"
NIL
NIL
```

The output shows that the target table has been saved in the file C:/Documents and Settings/Avri/foo.fsl, and can be loaded from there. If one does not wish to use the default parameters for the semantic space construction, the different components can be specified to the **analyse** function:

```
CL-USER 5 > (with-open-file (in "c:/ws.txt" :direction :input)
              (analyse in
                       8
                       '("black" "army" "death")
                       '("king" "queen" "fool")
                       "c:/testing.txt"
                       t))
;;; Compiling file c:/testing.txt ...
;;; Safety = 3, Speed = 1, Space = 1, Float = 1, Interruptible = 0
;;; Compilation speed = 1, Debug = 2, Fixnum safety = 3
;;; Source level debugging is on
;;; Source file recording is  on
;;; Cross referencing is on
; (TOP-LEVEL-FORM 1)
```

```
; (TOP-LEVEL-FORM 2)
; (TOP-LEVEL-FORM 3)
#P"c:/testing.fsl"
NIL
NIL
```

The first optional input is the window size which defaults to 10. The next two inputs are respectively the list of base words (also called context words) and the list of target words (i.e. words to be analysed). It is not expected that the user will directly write these list of words each time they are needed as, for example, the default list of context words contains 536 elements. Instead, the user should have a file containing all the words for one list, and use the **read-file** function to get the list, as in the following example where the output was omitted:

```
CL-USER 6 > (with-open-file (in "c:/ws.txt" :direction :input)
              (analyse in
                       8
                       (read-file "c:/basewords.txt")
                       (read-file "c:/targetwords.txt")
                       "c:/testing.txt"
                       t))
```

The final two inputs to this function are the file where to store the ***target*** table and the Gutenberg boolean. It is important to note that the target-words table is not saved under the exact file name as entered, but under the .fsl file with the same name, at the same location. In the above example, the table will be saved in "c:/testing.fsl". This anomaly will be removed from the program in the next version. Finally, the Gutenberg boolean specifies if the file analysed has been taken from the Gutenberg Project. This in turn specifies if the header that is present in all Gutenberg Project files must be stripped.

It is possible to use one of the other functions in order to analyse texts. If these functions do not save all the global variables, or the target table (depending on what is needed), then the user can do this manually using the functions **save-target-table** and **save-all-variables**. When saving all the variables, the target table can be normalised upon loading the file, whereas when saving only the target table one must make sure it is already normalised (as information required to normalise it at a later date is not saved). Both these functions only require the pathname of where to save the variables to in order to work, as the hash table is a global variable:

```
CL-USER 7 > (save-all-variables "c:/testing.txt")
;;; Compiling file c:/testing.txt ...
;;; Safety = 3, Speed = 1, Space = 1, Float = 1, Interruptible = 0
;;; Compilation speed = 1, Debug = 2, Fixnum safety = 3
;;; Source level debugging is on
;;; Source file recording is  on
;;; Cross referencing is on
```

```
; (TOP-LEVEL-FORM 1)
; (TOP-LEVEL-FORM 2)
; (TOP-LEVEL-FORM 3)
; (TOP-LEVEL-FORM 4)
; (TOP-LEVEL-FORM 5)
; (TOP-LEVEL-FORM 6)
#P"c:/testing.fsl"
NIL
NIL
```

Detailed explanation about how the other analysing functions work is not given in this document. The code and example programs (such as the **analyse-local-BNC** function) give general pointers as to how they are used. Once the ***target*** table is populated, or saved into a file, one can start observing the results of the analysis.

A.2 Querying the *target* table: observing the analysis results

If the ***target*** table has not been populated during the current session, the user must first load such a table from a file, before observing similarity between words. This operation is done using the **load** command in Lisp (or alternatively the **load-target-table** function which is a simple renaming of load) as such:

```
CL-USER 8 > (load "c:/testing.fsl")
; Loading fasl file c:\testing.fsl
#P"c:/testing.fsl"
```

If the ***target*** table has been saved with all the global variables, and needs to be normalised, this can also be done. The default normalisation function provided is the log odds-ratio. The function **finalize-target-table** takes care of this.

Once the ***target*** table has been normalised, similarity values for pairs of words can be calculated using the **similarity** function as shown in the following examples:

```
CL-USER 9 > (similarity "black" "white")
0.7966215831073316

CL-USER 10 > (similarity "black" "dog")
0.35816217445260423

CL-USER 11 > (similarity "good" "yellow")
NIL

CL-USER 12 > (similarity "man" "man")
1.0000000000000004
```

These examples are also a good illustration of what happens in special cases. When one of the inputs to the function is not in the ***target*** table, the function returns nil (as in interaction number 11). Finally, the precision of the calculations are shown by interaction 12. The similarity between a word and itself should be 1, therefore there is an error rate of about 0.0000000000000004 in that interaction. As a rule of thumb, it is useful to take into account only the first few digits when comparing similarity values. For example we can say that the similarity between "black" and "white" is about 0.800 and the similarity between "black" and "dog" is 0.358.

In order to replicate the different experiments, other functions to query the ***target*** table in different ways have been written. Most of these use the **similarity** function seen above, or optimise it for specific uses (such as not calculating the vector form of each word more than once...).

A.3 Further information

This appendix only gave a brief overview of the capabilities of the text analysing tools which were developed for this project. If one wishes to be able to use, adapt and extend the tools basic knowledge of Lisp is required. For any additional information about the program which cannot be derived from the examples or the source code, it is possible to email the author.

Appendix B

Code printout

The code written for this project is divided into files based on the uses of the functions. One file groups together the text analysis related functions (semal.lisp), another one groups the vector related functions (vect.lisp). The log odds-ratio functions are also grouped in a file (log-odds.lisp). Finally the user interface and its related functions are grouped together (in semalUI.lisp). The last file contains all the functions used to replicate priming, mediated priming and the IAT experiments. Every main function contains a commented header specifying the uses of the function, its input, its output and the functions that it uses. In addition to this a documentation string has been provide for each function. This documentation string is most useful in order to get the basic information of each function quickly while programming. In order to view the documentation string, while interacting with the Lisp Listener, it is sufficient to evaluate: (documentation #'X 'function), where X is the function's name.

The following is the printout of the code used for this project separated into the different files. Some of the formatting may have been slightly altered in order to fit the page margins.

B.1 semal.lisp: The text analysing functions

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Takes care of all the 'window' related functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;; function list: mkwindow, mv-one
;;;;
;;;;
;;;; function returning a 'double sided window' to analyse a text (x
;;;; words on each side)
;;;; INPUT: size of the window (of one side of the word)
;;;;        stream that will be read from to initialise it
;;;; OUTPUT: a window half full with the stream, ready to be used.
;;;; USES: mv-one, getword
(defun mkwindow (size stream)
```

```
  "makes a double sided window and fills it with words from the stream,
until the middle element is not nil"
  (let* ((winsize (1+ (* size 2)))
         (window (make-list winsize)))
    (dotimes (i (1+  size))
      (setf window (mv-one window (getword stream))))
    window))

;;;; moves the window forwards, given the next word in the text
;;;; INPUT: the current window and the next word int he text
;;;; OUTPUT: the new window (old one moved forward by one word)
(defun mv-one (window element)
  "moves a window one word forward: i.e. adds the element to the end
of the window and cuts one word off the other end"
  (rplacd (last window 2) ())
  (push element window))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; functions relating to reading from files or streams
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;;
;;;; function list: getword, getword-aux, read-file, wordlist, mkword
;;;;                strip-guten-header
;;;;
;;;;
;;;; reads from a stream and returns the first word
;;;; stream is an input stream and word is a list (enpty at the beginning)
;;;; INPUT: stream: a stream to read from
;;;;        word: the current word (as we read char by char)
;;;; OUTPUT: the first word of the stream
;;;; USES: getword-aux
(let ((words nil)) ;where we store all the words from one line of text
  (defun getword (stream)
    "returns a single word from a stream"
    (cond ((null words) (setf words (getword-aux stream))
           (if (null  words) nil (pop words)))
          (t (pop words)))))

;;;; aux function. reads from the stream until get a line which is not empty
;;;; or an end of file
;;;; USES: wordlist, mkword
(defun getword-aux (stream)
  (let ((line (read-line stream nil :eof))) ; get the line from the stream
    (cond ((equal :eof line) nil) ;detect end-of-file
      (t (setf line (remove-if (lambda (x) (OR
                                            (string= "" x)
                                            (eq #\< (char x 0))))
                               ;remove html and ""
                               (mapcar #'mkword (wordlist line))))

       (if (null line) (getword-aux stream)
         line)))))

;;;; reads a file and outputs a list of words, containing no punctuation
;;;; INPUT: a file
```

```
;;;; OUTPUT: the list of words (strings) contained in that file
;;;;         (with no punctuation and no "")
;;;; USES: wordList
(defun read-file (filename)
  "reads an entire file, and returns a list of words, with punctuation."
  (if (string= "" filename) nil ;checks if filename is empty and returns nil
    (with-open-file (infile filename)
      (do
          ((result () (append result (wordlist next)))
           (next (read-line infile nil) (read-line infile nil)))
          ((eq next nil) ;stop when no next line
           (remove-if (lambda (x) (string= "" x)) result))))))


;;;; Makes a list of words out of a string.(i.e. splits a string at spaces)
;;;; INPUT: a string
;;;; OUTPUT: a list of all the words contained in the input string.
;;;;         punctuation, space and numbers are removed (apart from -)
(defun wordlist (string)
  "tokenises a into a list of words (elements seperated by space).
multiple spaces are represented as empty strings)"
  (loop for i = 0 then (1+ j)
        as j = (position #\Space string :start i)
        collect (subseq string i j)
        while j))

;;;; function used to remove punctuation from strings, and turn it to lowercase
;;;; INPUT: a string
;;;; OUTPUT: a lowercase string equivalent to the input with all
;;;; spaces, numbers and punctuation removed (- allowed)
(defun mkword (word)
  "removes punctuation from a string"
  (let ((punct '(#\: #\; #\' #\@ #\. #\‘ #\ #\ #\, #\/ #\? #\# #\~
#\] #\[ #\{ #\} #\" #\! #\ #\$ #\% #\^ #\& #\* #\( #\) #\| #\\
#\_ #\ #= #\+ #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0 #\Space)))
    (string-downcase (remove-if (lambda (x) (member x punct)) word))))


;;;; function which reads the whole 'gutenberg project' file header
;;;; and ignores it
;;;; INPUT: a stream of a gutenberg project file
;;;; OUTPUT: t (and the header is not in the stream anymore)
;;;; may output an error if *END* is never found in the file
(defun strip-guten-header (stream)
  "given a stream made form a 'Gutenberg project' text, strips it's header"
  (do ((line "" (read-line stream)))
      ((search "*END*" line) t)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; specific functions for the initialisation,
;;;; finalization of the analysing functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;; function list: init-occurence, init-target-table, read-target-table,
;;;; save-target-table, finalize-target-table, save-all-variables
;;;;
```

```
;;;; initialises the general word-count hash table with the words
;;;; (given as strings)
;;;; INPUT: table: an empty hash table
;;;;        words: a list of words (strings) with which to initialise
;;;;               the hash-table
;;;; OUTPUT: a hash table with all the values associated with the given words
;;;;         initialised to 0
(defun init-occurence (table words)
  "initalises a hashtable with a set of strings as keys and the
value 0 associated with them"
  (dolist (w (remove-if (lambda (string) (string= "" string)) words)
            table)
    (setf (gethash w table) 0)))

;;;; initialises the target hashtable
;;;; INPUT: target-table: a hashtable to initialise
;;;;        target: the list of target words to initialise the target
;;;;                table with
;;;;        basewords: list of basewords (used as keys for each target
;;;;                   word's hastable)
;;;;        target and basewords must be a list of strings
;;;; OUTPUT: the initialised target table
;;;; USES: init-occurence
(defun init-target-table (target-table target basewords)
  "initialises the target table (hashtable with the targetwords associated
to simple hashtable)"
  (dolist (word target t)
    (setf (gethash word target-table)
          (init-occurence (make-hash-table :test #'equal)  basewords))))

;;;; loads a target table file (.fsl file) This is just a renaming of
;;;; the load function
;;;; INPUT: the path to the file containing the target-table/other
;;;; variables initilaised
;;;; OUTPUT: nothing, but the variables are loaded into the memory
(defun read-target-table (pathname)
  (load pathname))

;;;; function that stores in persisting memory the target hashtable
;;;; INPUT: the pathname where the .fsl file containing the
;;;; target-table will be created
;;;; USES: *target*
;;;; NOTE: leaves a residual file with the exact pathname as specified
;;;; by the user
(defun save-target-table (pathname)
  "stores only the target-table in a .fsl file, having the specified name
  (.fsl is automatically added to the pathname) "
  (with-open-file (out pathname
                       :direction :output
                       :if-does-not-exist :create
                       :if-exists :supersede)

    (dolist (x (coerce "(setq  *target* #.*target*)" 'list) t)
      (write-char x out)))
  (compile-file pathname))

;;;; finalises values in the hashtable (applies the normalisation function)
```

```
;;;; INPUT: the window size
;;;; OUTPUT: nothing (works by side-effects)
;;;; USES: *target* *word-count* and *corpus-length*
;;; NOTE: in the next version of the program must make sure that
;;;; *winsize* is consistently used in all the program.
;;;; Thus the function will not have to take winsize as an input.
(defun finalize-target-table (winsize)
  "applied the log-odds-ratio to the *target* table"
  (map-log-odds-ratio *corpus-length* winsize *target*))

;;;; saves all the global variables (*target* *winsize* *word-count*
;;;; and *corpus-length*)
;;;; to a file
;;;; INPUT: the pathname where the .fsl file containing the global
;;;;        variables will be created
;;;; NOTE: there must be a simpler way to write the #. command to a file,
;;;;       but all other results failed (due to lack of knowledge?).
(defun save-all-variables (pathname)
  "saves *target* *winsize* *word-count* and *corpus-length* to
a specified .fsl file"
  (with-open-file (out pathname
                       :direction :output
                       :if-does-not-exist :create
                       :if-exists :supersede)
    (dolist (x (coerce "(setq  *target* #.*target*)
(setq *word-count* #.*word-count*)
(setq *corpus-length* #. *corpus-length*)
(setq *winsize* #.*winsize*)" 'list) t)
      (write-char x out)))
  (compile-file pathname))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; the basic text analysing functions (and auxiliaries)
;;;; these are used as the main building blocks for the
;;;; analysing functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;; function list: read-one, update, middle
;;;;

;reads a word and depending on the middle word of the window
;updates the cooccurence counts
;USES: update, *word-count*
;;;; updates the word co-occurence count of the middle word of the
;;;; current window
;;;; in addition to the simple occurence count
;;;; INPUT: window: the current window to analyse
;;;;        window-size: the total size of the window
;;;;        target: the target words hashtable
;;;;        basewords: the list of the basewords
;;;; OUTPUT: works by side effect
;;;; USES: update, *word-count*
;; WE ASSUME NO TARGET-WORD IS A BASEWORD
(defun read-one (win winsize target &optional (word-count *word-count*))
  "updates the occurrence and co-occurence values of the middle word of
the current window"
```

```
    (let* ((mid (middle win winsize)) ;store middle of the window
         (wc-mid (gethash mid word-count))
         (t-mid (gethash mid target)))
      (declare (optimize speed))
; if appropriate increment w-c table
      (when wc-mid (incf (gethash mid word-count)))
      (when t-mid (update t-mid win))))


;;;; auxiliary function for read-one
;;;; updates the wordcounts of relevant words in a list
;;;; INPUT: table: hashtable of word-count (must be initialised)
;;;;        words: a list of words taken from a text to analyse
;;;; OUTPUT: the original list of words
(defun update (table words)
  (mapcar (lambda (w) (when w ;when the word is not nil
                        (when (gethash w table) ;when the word is baseword
                          (incf (gethash w table)))))
          words));mapc instead?


;;;; function that finds the middle element of a window
;;;; INPUT: a window and it's size (the size of one side)
;;;; OUTPUT: the middle element
(defun middle (window win-size)
  "returns the middle element of a list (given the list and its size)"
  (nth win-size window))




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; The Analysing functions (all different versions)
;;;; and the global variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;; variable list: *target* *word-count* *winsize*
;;;;
;;;; function list:


;;; the global hashtables
(defvar *target* (make-hash-table :test #'equal))
(defvar *word-count* (make-hash-table :test #'equal))
;;; the windowsize (mostly used for the GUI)
(defvar *winsize* 10)
;;; the setter for the winsize
(defun set-winsize (num)
  (setq *winsize* num))

;;;; Loading the files needed for the analysing functions
;;; loading vector related functions
(load "vect.lisp")
;;; loading log-odds-ratio functions
(load "log-odds.lisp")
```

```
;;;; small functions that return the basewords and the targetwords
;;;; for the text analysis
;;;; NOTE: Must make sure that the path to the files is correct when
;;;;       using on a different computer

;;;; INPUT: nil
;;;; OUTPUT: the list of the default basewords as a list of strings
;;;; USES: read-file
(defun get-basewords ()
  "returns the set of default basewords"
  (mapcar #'string-downcase (read-file "c:/lowe-basewords.txt")))

;;;; INPUT: nil
;;;; OUTPUT: the list of the default targetwords as a list of strings
;;;; USES: read-file
(defun get-targetwords()
  "returns the set of default targetwords"
  (mapcar #'string-downcase
         (append  (read-file "c:/targetwords.txt")
                  (read-file "c:/priming-target-words.txt"))))


;;;; the main function that analyses the corpus (read in from the stream)
;;;; Maybe need to passe basewords and targetwords when call the function
;;;; INPUT: a stream to read from
;;;; OUTPUT: nil, but the target hashtable is updated with the
;;;; co-occurence counts and the odds ratio applied to it
;;;; USES: init-target-table, mkwindow, mv-one, getword,
;;;;       map-log-odds-ratio, read-one, strip-guten-header
(defun analyse (stream
                &optional (winsize 10)
                (basewords (get-basewords))
                (targetwords (get-targetwords))
                (pathname "foo.txt")
                (gutenberg nil))
  (clrhash *target*)(clrhash *word-count*)
  (init-target-table *target* targetwords basewords)
  (init-occurence *word-count* (append targetwords basewords))
  (when gutenberg (strip-guten-header stream))
  (do
      ((corpus-length 0 (1+ corpus-length))
       (win (mkwindow winsize stream) (mv-one win (getword stream))))
      ((null (nth winsize win));stop when getword returns null
       ;apply odds ratio to target,at the end
       (progn (map-log-odds-ratio corpus-length winsize *target*) corpus-length
         (save-target-table pathname)))

    (declare (optimize speed))
    (read-one win winsize *target*)))


;;;; analyses a list of files (given in a pathnames)
;;;; INPUT: a list of pathnames (files) to analyse
;;;;        optionally the size of the window, the set of base and target words
;;;;        and a pathname to save the variables to (if nil then don't save)
;;;; OUTPUT: *target* and *word-count* table are populated
;;;; USES: init-target-table, init-occurence, *target*, *word-count*,
```

```
;;;;        map-log-odds-ratio.
;;;;       save-target-table, parse-file
(defun analyse-files (file-list
                      &optional (winsize 10)
                      (basewords (get-basewords))
                      (targetwords (get-targetwords))
                      (pathname "foo.txt"))

  (init-target-table *target* targetwords basewords)
  (init-occurence *word-count* (append targetwords basewords))
  (let ((corpus-length 0))
    (dolist (file file-list t)
      (setf corpus-length (+ corpus-length
                             (parse-file file winsize))))
    (map-log-odds-ratio corpus-length winsize *target*)
    (when pathname (save-target-table pathname))))


;;;; auxiliary function for analyse-all
;;;; analyses a single file and returns it's length
(defun parse-file (name winsize)
  (with-open-file (stream name :direction :input)

    (do ((file-length 0 (1+ file-length))
         (win (mkwindow winsize stream) (mv-one win (getword stream))))
        ((null (nth winsize win)) file-length)
      (read-one win winsize *target*))))



;;;; analyses some files, but does not finalise it.
;;;; hashtables: *target* and *word-count* as well as integer
;;;; *corpus-length* and *winsize* can all be saved to file;
;;;;  assumes global vars already initialised
;;;; INPUT: the file list to analyse and optionally: the window size,a
;;;; pathname to save the variables to (only if last line uncommented),
;;;; an optional file to load which is only to be used if a
;;;; computation is being resumed...
(defun analyse-part (file-list
                     &optional (winsize *winsize*)
                     (pathname "part.txt")
                     (file-to-load "part.fsl"))
  (when file-to-load (load file-to-load)) ;inits hashtables etc... if resuming
  (dolist (file file-list t)
    (setf *corpus-length* (+ *corpus-length*
                             (parse-file file winsize))))
  ;(save-all-variables pathname)
)



;;; just small program to analyse all the BNC

;;;analysing the bnc, in parts...
(defvar *corpus-length* 0)
```

```
;;;; function that analyses a local copy of the BNC, and saves checkpoints
;;;; INPUT: win: the size of the window
;;;;        swin: the size of the window as a string
;;;;              (used to name the file with global vars)
;;;; OUTPUT: *target* hashtable is populated with the relevant
;;;;          data, and must be normalised in order to carry
;;;;          out analysis on it
(defun analyse-local-BNC (win swin)
  (let ((dirs '("A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K"))
        (bwords (get-basewords))
        (twords (append (reduce #'append *ratcliff-words*)
                        (get-targetwords)
                        (read-file "c:/flowers.txt")
                        (read-file "c:/instruments.txt")
                        (read-file "c:/weapons.txt")
                        (read-file "c:/insects.txt")
                        (read-file "c:/pleasant.txt")
                        (read-file "c:/unpleasant.txt"))))
 ; clear global variables and initialise them
  (clrhash *target*)(clrhash *word-count*)
  (init-target-table *target* twords bwords)
  (init-occurence *word-count* (append twords bwords))
  (dolist (dir dirs t) ; for each A B C D ....
    (declare (optimize speed))
    (dolist (f (directory (concatenate 'string
                                       "C:/BNC/Texts/"
                                       dir "/"))t);for each subfolder A0
      (declare (optimize speed))
      (analyse-part (directory (namestring f))
                    win
                    (concatenate 'string "bnc-part" dir ".txt")
                    nil)
      (print (concatenate 'string "Done directory " (namestring f)))))
  (save-all-variables (concatenate 'string swin "bnc-partK.fsl")))
)



;;;;;;;;;;;;;;;;;;; vector related functions ;;;;;;;;;;;;;;;;;;;;;;
;;;; are not in a different file (with the others) only due to speed
;;;; considerations

;;;; gets the vector form out of the hashtable representation
;;;; of a targetword
;;;; NOTE: must set the basewords to be the correct list using
;;;; set-vector-basis if not using the default set of basewords.
(let ((basewords (get-basewords)))
  ; sets the targetwords which are used to get the vector form
  (defun set-vectors-basis (base)
    (setf basewords base ))

  ;; INPUT: a hashtable representing a targetword
  ;; OUTPUT: a vector form (list of numbers) representing the targetword as a
  ;;         vector/point in the semantic space
  (defun get-vector-form (htable)
    (mapcar #'(lambda (tword) (gethash tword htable)) basewords)))
```

```
;;;; calculates the similarity (cos) between two words given as strings...
;;;; INPUT: the two words to compare and optionally a target table from which
;;;;        to take their values
;;;; OUTPUT: the similarity (cosine) between the two words in
;;;; the semantic space
;;;; USES: cosine, get-vector-form, optionally *target*
(defun similarity (word1 word2 &optional (target *target*))
  "returns the similarity (cosine) between two target words"
  (unless (OR (null word1) (null word2) (string= "" word1) (string= "" word2))
    (let ((table1 (gethash word1 target))
          (table2 (gethash word2 target)))
      (cond ((OR (null table1) (null table2)) nil)
            (t (cosine (get-vector-form table1) (get-vector-form table2)))))))
```

B.2 log-odds.lisp: The log odds-ratio related functions

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; functions relating to the log-odds-ratio calculations
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;;
;;;; function list: odds-ratio, map-log-odds-ratio, map-log-odds-ratio-aux

;;;; INPUT: base: the base word we are interested in (as a string)
;;;;        target: the target word we are interested in (idem)
;;;;        target-table: hashtable of the target word
;;;;        corpus-size: the length of the corpus that has been analysed
;;;;        t-win-size: the total window size (usually 20)
;;;; OUTPUT: the odds-ratio for the given word pair as calculated by:
;;;;         proba of seeing base * proba of being in t's window
;;;;         where: proba of being in t's window is: num of win with
;;;; t/total num of windows
;;;; USES:  *word-count*
(defun odds-ratio (base target target-table corpus-size t-win-size)
  (let* ((wft (* (gethash target *word-count*) t-win-size))
         (wfb (* (gethash base *word-count*) t-win-size))
         (fbt (gethash base target-table))
         (wn (* corpus-size t-win-size))
         (fbnt (- wfb fbt))
         (fnbt (- wft fbt))
         (fnbnt (- wn fbnt fnbt fbt)))
        (/ (* fbt fnbnt) (* fbnt fnbt))))

;;;; maps the odds ratio calculation to all target words in the *target* table
;;;; INPUT: the size of the corpus and the size of the window
;;;; OUTPUT: the *target* table after application of odds ratio to all elements
;;;; USES: map-log-odds-ratio-aux, *target*
(defun map-log-odds-ratio (corpus-size window-size &optional (target *target*))
  (let ((total-window-size (* window-size 2)))
    (maphash ;map on to all targetwords hashtables
     #'(lambda (tword ttable)
         (maphash ;the application of logs odd ratio thing to all basewords
```

```
          #'(lambda (bword count)
              (if (zerop (gethash bword ttable))
                    nil
                    (when (> 0 (setf (gethash bword ttable)
                           (log (odds-ratio bword
                              tword
                              ttable
                              corpus-size
                              total-window-size) 10)))
                           ;truncating values < 0
                          (setf (gethash bword ttable) 0))))
          ttable))
     target)))
```

B.3 vect.lisp: The n-dimensional vector related functions

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; functions relating to vectors and vector arithmetic
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;;
;;;; function list: cosine, normalize, norm, dot

;;;; to calculate the similarity between two points in space

;;;; function which finds the cosine of the angle between 2 vectors
;;;; INPUT: the two vectors
;;;; OUTPUT: the cosine between the two vectors
;;;;         (calculated as the dot product of the two normalised vectors)
;;;; USES: dot, normalise
(defun cosine (vect1 vect2)
  "calculates the cosine between to n dimentional vectors
using the dot product of the normalized vectors"
  (dot (normalize vect1) (normalize vect2)))

;;;; normalises a vector (represented as a list of numbers)
;;;; INPUT: the vector to normalise
;;;; OUTPUT: the normalised vector (each of the coordinates divided by
;;;;         the norm)
;;;; USES: norm
(defun normalize (vect)
  "normalizes an n dimentional vector"
  (let ((vect-norm (norm vect)))
        (if (zerop vect-norm)
            vect
          (mapcar #'(lambda (x) (/ x vect-norm))
                  vect))))

;;;; calculates the norm/length of a geometric vector (represented as a list)
;;;; INPUT: the vector (list of numbers)
;;;; OUTPUT: the length/norm of the vector
;;;;         (square root of sum of the squares of coordinates)
(defun norm (vect)
```

```
  "calculates the norm of an n dimentional vector"
  (sqrt (reduce #'+
                (mapcar #'(lambda (x) (* x x)) vect))))

;;;; dot product of two vectors
;;;; INPUT: two vectors (lists) of the same dimentionality (length)
;;;; OUPTUT: the dot product of the two vectors
(defun dot (vect1 vect2)
  "calculates the dot product between two vectors"
  (reduce #'+ (mapcar #'* vect1 vect2)))
```

B.4 semalUI.lisp: The User Interface

```
;defining the package
(defpackage semal-user-interface
  (:add-use-defaults t)
  (:use "CAPI")
)

;using it
(in-package semal-user-interface)

;loading all the functions from semal
(load "semAl.lisp")


;; text input pane, takes the name of the corpus file to parse
(setq corpus (make-instance 'text-input-pane
                            :title "Target-table to load: "
                            :title-position :left
                            :text "bnc-partK.fsl"))

;; browse button for choosing a corpus
(setq corpus-but (make-instance 'button
                          :data "Browse..."
                          :callback
                          #'(lambda (&rest args)
                              (browse corpus "Choose target-table to load: "))))

;;; the load button, which loads the desired 'corpus' i.e. target-table
(setq load-but
      (make-instance 'button
                     :data "Load"
                     :callback
                     #'(lambda (&rest args)
                         (let ((filename (text-input-pane-text corpus)))
                           (when (load filename
                                       :if-does-not-exist nil)
                                 ;set the status-bar
                             (apply-in-pane-process
                                   status-report
                                   #'(setf title-pane-text)
                                   (format nil "~S loaded. window-size: ~D"
                                           filename
                                           cl-user::*winsize*)
```

```
                              status-report)
                        ;import the *target* hash-table into current package
                              (setq *target* cl-user::*target*)
                              (setq *word-count* cl-user::*word-count*)
                              (map-log-odds-ratio
                                cl-user::*corpus-length* cl-user::*winsize*)
                              (print *target*)
                              )))))


;; the status report pane, tells user when a target file has been loaded, etc..
(setq status-report (make-instance 'title-pane
                                   :title "Status: "
                                   :title-position :left
                                   :text
                                    "No File Loaded. Please Load a .fsl file"))


;; the row containing the load button and the status reporting title-panes
(setq load-row (make-instance 'row-layout
                              :description (list load-but status-report)))

;; the row with the corpus text area and the browse button
(setq c-row (make-instance 'row-layout
                           :description (list corpus corpus-but)
                           :gap 10))



;; an input for any additional info for the target-words (num of pairs
;; or num of words to compare with only one word)
;; not giving it a title or a value as it depends on what option is
;; set in tw-options
(setq tw-num (make-instance 'text-input-pane
                            :title "targetword file: "
                            :text "c:/targetwords.txt"
                            :max-characters 45
                            :internal-min-width '(character 44)
                            :internal-max-width '(character 45)))


;; the target-word apply button (visible only if applicable)
(setq target-but (make-instance 'button
                                :data "Browse..."
                                :callback
                                #'(lambda (&rest args)
                                          (target-button-action
                                           tw-num
                                           (choice-selection tw-options)))))


;; the target-words options
(setq tw-options (make-instance 'radio-button-panel
                                :title "Targetwords selection:"
                                :items (list "Load targetword file"
                                             "Compare targetwords pairwise"
                                             "Compare targetwords manually")
```

```
                                :layout-class 'column-layout
                                :selection-callback #'(lambda (sel &rest args)
                                                        (target-selection sel))))


;; the row layout for the right side of the tw-selection
(setq t-opt-row (make-instance 'row-layout
                                    :description (list tw-num target-but)))

;; the row layout for all the target options
(setq t-row (make-instance 'row-layout
                           :description (list tw-options t-opt-row)))

;; the similarity display columns (initially empty)
(setq word1 (make-instance 'column-layout
                           :description nil
                           :title ""))
(setq word2 (make-instance 'column-layout
                           :description nil
                           :title ""))
(setq sim (make-instance 'column-layout
                           :description nil
                           :title ""
                           :gap 9))


;; the row layout for the similarity display part of the screen
(setq similarity-row (make-instance 'row-layout
                                    :description (list word1 word2 sim)))

;; the ok button
(setq ok-but (make-instance 'button
                            :data "Analyse"
                            :callback
                            #'(lambda (&rest args)
                                (if (= (choice-selection tw-options)
                                       0)
                                    (write-similarity-file)
                                  (disp-similarity)))))

;;the exit button
(setq exit-but (make-instance 'button
                              :data "Exit"
                              :callback
                              #'(lambda (&rest args)
                                  (destroy win))))

;; the ok and exit buttons on a row
(setq but-row (make-instance 'row-layout
                           :description (list ok-but exit-but)
                           :x-adjust :right))

;; main window
(setq win (make-instance 'interface
                         :visible-min-width 500
                         :title "Corpus Parsing"
                         :layout
```

```
                        (make-instance 'column-layout
                                       :description
                          (list c-row load-row t-row similarity-row but-row)
                                       :gap 5
                                       :internal-border 5)))

;; display the user interface
(display win)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; functions specifying behaviour of GUI
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun check-inputs (window-size basewords targetwords stream)
  (if (OR (null window-size)
          (null basewords)
          (null targetwords)
          (string= stream "")) nil t))


;;;; function which returns the cosine between the two targetwords
;;;; (with some text)
;;;; INPUT: nothing
;;;; OUTPUT: a string containing the cosine between two target words
;;;;         (as written in the GUI)
(defun similarity-display ()
  (let ((v1 (get-vector-form
             (gethash (text-input-pane-text target1-disp) *target*)))
        (v2  (get-vector-form
              (gethash (text-input-pane-text target2-disp) *target*))))
    (format nil "The similarity between the two words is:~\%~,3F"
            (cosine v1 v2))))



;;;; behaviour of the target radio buttons
;;;; (and their effect on the rest of the GUI)
(defun target-selection (selection)
  (let ((btext "") (title "Targetword file: ")(itext "c:/targetwords.txt")
        (w1-title "First word")(w2-title "Second word")
        (sim-title "Similarity"))
    (cond ((string= "Load targetword file" selection)
           (setf btext "Browse...")
           (setf w1-title "") (setf w2-title "") (setf sim-title ""))
          ((string= "Compare targetwords pairwise" selection)
           (setf btext "Apply")
           (setf title "Number of targetwords [1-20]: ")
           (setf itext "3"))
          ((string=  "Compare targetwords manually" selection)
           (setf btext "Apply")
           (setf title "Number of targetwords [1-20]: ")
           (setf itext "3")))
    ;setting the button text
    (apply-in-pane-process target-but
                           #'(setf item-text)
                           btext
                           target-but)
```

```
    ;setting the text-pane title
    (apply-in-pane-process tw-num
                           #'(setf titled-pane-title)
                           title
                           tw-num)
    ;empty similarity row's components
    (apply-in-pane-process similarity-row
                           #'(lambda (list)
                               (mapcar (lambda (layout)
                                         (setf (layout-description
                               layout)
                                               nil))
                                       list))
                           (layout-description similarity-row))
    ;clear text-pane's text
    (apply-in-pane-process tw-num
                           #'(setf text-input-pane-text)
                           itext
                           tw-num)
    ;deletes the titles on all three columns (word1 word2 sim)
    (apply-in-pane-process similarity-row
                           (lambda (l1 l2)
                                   (mapcar #'(setf titled-pane-title)
                                           l1 l2))
                           (list w1-title w2-title sim-title)
                           (layout-description similarity-row))
     ))

;; is either used as a 'browse' button or as a 'apply changes' button
;; depending on the value of the radio-buttons
(defun target-button-action (tw-num selection)
  (if (= selection 0) (browse tw-num)
    (let ((number (read-from-string (text-input-pane-text tw-num))))
      ; should inform user of this (next version)
      (unless (AND (< 0 number) (< number 20)) (setf number 20))
      (make-results-display number (= selection 1)))))


;; function which creates three columns of text-inputs
;; labelled "word 1" "word 2" and "similarity"
;; the first input determins the number of rows in the columns
;; if the second input is nil then the first column only has one row
(defun make-results-display (num pairwise)
  (let ((columns (if pairwise (list word1 word2 sim) (list word2 sim)))
        (type nil ))
    (dolist (col columns)
      (setf type (if (eq sim col) 'title-pane 'text-input-pane))
    ; empty whatever they contain before
      (apply-in-pane-process col
                             #'(setf layout-description)
                             nil
                             col)
;fills col with the num of input-text-panes specified
      (dotimes (x num nil)
        (apply-in-pane-process col
                               #'(setf layout-description)
                               (cons (make-instance type);'text-input-pane)
```

```
                                    (layout-description col))
                                col))
      )
    )
  (when (not pairwise)
        (apply-in-pane-process word1
                               #'(setf layout-description)
                               (list (make-instance 'text-input-pane))
                                word1)))


;; the browse function: given a input-text-pane
;; it copies the selected file from browsing into the text-pane
(defun browse (in-text &optional (title "Choose file to load: "))
  (let ((file (prompt-for-file title :pathname "c:/")))
    (unless (null file)
      (apply-in-pane-process
       in-text
       #'(setf text-input-pane-text)
       (namestring file) in-text))))

(defun concat-s (list &optional (res ""))
  (cond ((null list) res)
        (t (concat-s (cdr list) ;concat the string
                     (concatenate 'string res (car list))))))

;;;; function which extracts basewords from a target-hashtable
;;;; basewords come as a set of symbols
;;;; USES: *target*
(defun extract-basewords ()
  (let ((basewords ()))
    (maphash (lambda (tword tables)
                (unless basewords
                  (maphash (lambda (bword val)
                             (setq basewords (cons bword basewords)))
                           (car tables))))
             *target*)
      basewords))

;;;; extracts the target-words from the target table
;;;; USES: *target*
(defun extract-targetwords ()
  (let ((targetwords ()))
    (maphash (lambda (tword tables)
                     (setq targetwords (cons tword targetwords)))
             *target*)
    targetwords))

;;;; does not use 'similarity' function directly for efficiency reasons
;;;; writes-pairwise similarity or a big number of targetwords, to a file.
(defun write-similarity-file ()
  (with-open-file (out (concatenate 'string
                                    (string-right-trim ".txt"
                                      (text-input-pane-text tw-num))
                                    "-sim.csv")
                       :direction :output
                       :if-does-not-exist :create
```

```
                    :if-exists :supersede)
    (let* ((targetwords (read-file (text-input-pane-text tw-num)))
           (symbols (mapcar #'string-downcase targetwords))
           (vectors ; vector form of each targetword to analyse
            (mapcar #'(lambda (tword) (get-vector-form
                                       (gethash tword *target*))) symbols))
           (vect-hashtable (make-hash-table :size (length symbols)))
           (output ""))
      (do ((symb symbols (cdr symb))
           (vect vectors (cdr vect)))
          ((null symb) t)
              ;intialisating the vect-hashtable
        (setf (gethash (car symb) vect-hashtable) (car vect)))

      (dolist (tword symbols t)
        (dolist (ttword (cdr (member tword symbols)) t)
          (setf output
             (concatenate 'string output
                        (format nil "~S,~S,~S"
                                tword ttword
                                (cosine (gethash tword vect-hashtable)
                                        (gethash ttword vect-hashtable)))
                            '(#\Newline)))))

      (write-string (remove-if #'(lambda (x) (equal #\" x)) output)
                    out))))


;;;; displays the similarity values of the words in the text panes, on
;;;; the titled-pane
(defun disp-similarity ()
  (let ((w1-disp (layout-description word1))
        (w2-disp (layout-description word2))
        (sim-disp (layout-description sim)))

    (do ((w2 w2-disp (cdr w2))
    ;steps w1 only if applicable
         (w1 w1-disp (if (null (cdr w1)) w1 (cdr w1)))
         (sim sim-disp (cdr sim)))
        ((null w2) t);stop looping when w2 is null
      (apply-in-pane-process (car sim)
                             #'(setf title-pane-text)
                             (format nil "~,4F"
                               (similarity (text-input-pane-text (car w1))
                                           (text-input-pane-text (car w2))
                                     *target*))
                             (car sim)))))
```

B.5 expReplication.lisp: Replicating the experiments

```
;;;; set of functions used to output results of similarity between
;;;; words which are mediated
;;;; or related primes. Following broch and loca's paper.
```

```
;;;; The documentation is not complete for these functions as they are
;;;; only displayed here to illustrate how to use the tools developed
;;;; by the author.


;;;; global variable: all the words for the mckoon-ratcliff experiments
(setq *ratcliff-words* '(("baby" "room" "hospital" "child")
("kids" "father" "young" "children")
("knife" "putty" "kitchen" "blade")
("sky" "fireworks" "night" "blue") ("wave" "radio" "heat" "brain")
("floor" "manufacturer" "convention" "ceiling")
("town" "flames" "residents" "city")
("nurse" "public" "army" "doctor")
("ground" "stake" "earthquake" "earth")
("plant" "growers" "power" "grow")
("shoe" "workman" "textile" "foot") ("leg" "amputation" "left" "arm")
("cake" "candles" "piece" "bake") ("girl" "love" "death" "boy")
("trucks" "sound" "fire" "cars")
("nation" "conscience" "newspapers" "country")
("pie" "cream" "apple" "crust") ("mind" "image" "doubt" "memory")
("grass" "plane" "acres" "green") ("hand" "guard" "cash" "finger")
("wound" "blood" "bullet" "heal")
("home" "morning" "vacation" "house")
("woman" "affair" "police" "man")
("letters" "protest" "calls" "numbers")
("games" "season" "war" "play")
("church" "mainstream" "separation" "priest")
("light" "glass" "sales" "lamp") ("sleep" "days" "hours" "bed")
("food" "flowers" "emergency" "stomach") ("water" "holes" "air" "ocean")
("window" "rain" "bedroom" "door") ("law" "welfare" "state" "justice")
("tree" "branch" "family" "leaf") ("stars" "female" "movie" "moon")
("song" "show" "theme" "music")
("crowd" "candidate" "cheering" "people")
("ship" "transport" "passenger" "porthole")
("health" "package" "public" "sickness")
("army" "protest" "officer" "soldier")
("smoke" "passenger" "black" "tobacco")))


;;;; global variable: all the words for the balota-lorca experiments
(setq *balota-words*
'(("stripes" "tiger" "lion") ("box" "sand" "beach")
("quiet" "peace" "war") ("pie" "cake" "birthday")
("vegetable" "animal" "deer") ("bubbles" "blow" "breeze")
("necklace" "pearl" "oyster") ("smell" "nose" "eyes")
("glass" "hour" "minute") ("drink" "water" "soap")
("bell" "church" "priest") ("carpet" "floor" "ceiling")
("kick" "foot" "hand") ("bounce" "ball" "bat")
("sweet" "sour" "lemon") ("color" "blue" "sky")
("cotton" "soft" "hard") ("bean" "coffee" "tea")
("letter" "number" "phone") ("lawyer" "doctor" "nurse")
("island" "fantasy" "reality") ("trigger" "gun" "knife")
("dance" "square" "circle") ("turtle" "slow" "fast")
("cheese" "mouse" "cat") ("snow" "winter" "summer")
("finger" "ring" "wedding") ("hair" "brush" "tooth")
("glove" "baseball" "sport") ("silk" "smooth" "rough")
("bottle" "baby" "cry") ("milk" "cow" "bull")
```

```
("syrup" "maple" "tree") ("lead" "pencil" "pen")
("grape" "wine" "beer") ("dark" "night" "day")
("clock" "watch" "wrist") ("coal" "black" "white")
("tank" "army" "navy") ("duckling" "ugly" "pretty")
("hot" "sun" "moon") ("knob" "door" "window")
("stop" "bus" "school") ("peak" "mountain" "valley")
("slick" "oil" "gas") ("thorn" "rose" "flower")
("feather" "light" "heavy") ("collar" "shirt" "pants")))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; For the Balota-lorch experiments
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-related (prime-list)
  "function that gets the related word of a certain target word in the
*word* list"
  (cadr prime-list))

(defun get-mediated (prime-list)
  "function that gets the mediated word of a certain target word
in the *word* list"
  (caddr prime-list))


;;code the get unrelated that returns a number of unrelated words....
(defun get-unrelated (tword n &optional (words *balota-words*))
  "function that gets n unrelated words for a certain target word
in the *word* list"
  (if (> n (1- (length words))) nil
    (let ((counter n) (unrelated ())
          (words (remove-if (lambda (x) (equal (car x) tword)) words)))
      (do ((element (cadr (nth (random (length words)) words))
                    (cadr (nth (random (length words)) words))))
          ((zerop counter) unrelated)
        (unless (find element unrelated :test #'equal)
          (decf counter)
          (push element unrelated))))))

(defun get-all-targets ()
  "returns the list of all the target words found in *balota-words*"
  (mapcar #'car *balota-words*))


;replicates the balota experiment
(defun balota ()
  "Outputs to c:/balota.csv the statistics replicating the
balota-lorch (1986) experiments. To best view the results,
import the file to excel and calculate the average of each column"
  (with-open-file (results "c:/balota.csv"
                           :direction :output
                           :if-exists :supersede)
    (dolist (primes *balota-words* t)
      (let ((tw (car primes)))
      (format results (concatenate 'string tw ",~,3F,~,3F,~,3F~\%" )
              (similarity tw (get-related primes) *target*)
```

```
              (similarity tw (get-mediated primes) *target*)
              (average (mapcar #'(lambda (x) (similarity tw x *target*))
                               (get-unrelated tw 20))))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; For the McKoon-Ratcliff experiment
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



(defun get-freePrime (prime-list)
  "returns a free prime given a specific *ratcliff-words* quadruple"
  (car (last prime-list)))

(defun get-highprime (prime-list)
  "returns a high t value prime given a specific *ratcliff-words* quadruple"
  (caddr prime-list))

(defun get-lowprime (prime-list)
  "returns a low t value prime given a specific *ratcliff-words* quadruple"
 (cadr prime-list))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; returns all the primes and the target for a specific target
;(in one of the lists of words)
(defun get-all-primes (target list)
  "returns all the prime words to a specific target word.
These are found in the list supplied
(usually *balota-words* or *ratcliff-words*)."
  (find target list :test #'(lambda (x y) (equal x (car y)))))
; if use this version, must also change the get- functions accordingly
; (take out the search)


;; ratcoon replication
(defun ratcliff ()
  "Outputs to c:/ratcliff.csv the statistics replicating the
McKoon-ratcliff (1992) experiments. To best view the results,
import the file to excel and calculate the average of each column"
  (with-open-file (out "c:/ratcliff.csv" :direction :output
                                         :if-exists :supersede)
   (dolist (primes *ratcliff-words* t)
     (let ((target (car primes)))
       (format out (concatenate 'string target ",~,3F,~,3F,~,3F,~,3F~\%")
             (similarity target (get-freePrime primes) *target*)
             (similarity target (get-highPrime primes) *target*)
             (similarity target (get-lowPrime primes) *target*)
             (average (mapcar #'(lambda (x) (similarity target x *target*))
                        (get-unrelated target 10 *ratcliff-words*))))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; IAT test replication functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun get-pleasant ()
  (read-file "c:/pleasant.txt"))

(defun get-unpleasant ()
  (read-file "c:/unpleasant.txt"))

(defun get-flowers ()
  (read-file "c:/flowers.txt"))

(defun get-insects ()
  (read-file "c:/insects.txt"))

(defun get-instruments ()
  (read-file "c:/instruments.txt"))

(defun get-weapons ()
  (read-file "c:/weapons.txt"))


;;;; replicated IAT test (compare words with pleasantness and unpleasantness)
;;;; only looks at words which have occured more than 100 times in the text...
;;;; INPUT: wlist: the word list that we wish to compare to
;;;;               pleasant/unpleasant words
;;;;        cat-name: the category name used to name the output file
;;;; OUTPUT: a .cvs file that should be imported into excel
;;;; USES: average, *word-count*, similarity, get-pleasant, get-unplesant
(defun iatexp (wlist cat-name)
  (with-open-file (out (concatenate 'string
                                    "c:/IATexp-" cat-name ".csv")
                       :direction :output
                       :if-exists :supersede)
    (dolist (flow
             (remove-if #'(lambda (w) (< (gethash w *word-count*) 100))
                        wlist)
             t)
      (format out "~D~C~,3F~C~,3F~\%"
              (gethash flow *word-count*)
               #\Tab
              (average (mapcar #'(lambda (x) (similarity flow x))
                               (get-pleasant)))
              #\Tab
              (average (mapcar #'(lambda (x) (similarity flow x))
                       (get-unpleasant)))))))


;compares the word with pleasant and unpleasant words and outputs the
;two values...
(defun good-or-bad (word)
  (list
   (list "pleasant"(average (mapcar #'(lambda (x)(similarity word x))
                            (get-pleasant))))
   (list "unpleasant"(average (mapcar #'(lambda (x)(similarity word x))
 (get-unpleasant))))))
```

```
; calculates the average of a list of numbers
(defun average (numlist)
  "calculates the average of a list of numbers"
  (unless (null (car numlist)) (/ (reduce #'+ numlist) (length numlist))))

; gives values to how pleasant and unpleasant a word is
(defun quick-test (wlist)
  (let ((flow (mapcar
               #'good-or-bad
               (remove-if #'(lambda (w) (< (gethash w *word-count*) 100))
                          wlist))))
    (list
     (list "pleasant" (average (mapcar #'cadar flow)))
     (list "unpleasant" (average (mapcar #'cadadr flow))))))


; once the file has been analysed, outputting the results, etc...
;;;; outputs the results in a format readable by the CCVISU tool
;;;; (http://mtc.epfl.ch/~beyer/CCVisu/
;;;; (in order to produce force directed graphs)
(defun exploring (name)
  (let ((twords (remove-if #'(lambda (x) (< (gethash x *word-count*) 20))
                           (read-file "c:/targetwords.txt"))))
    (with-open-file (out (concatenate 'string "c:/" name ".rsf")
                         :direction :output
                         :if-does-not-exist :create
                         :if-exists :supersede)
      (dolist (w twords t) ;foreach word from our targetwords
        ;no multiple comparisons between same pairs
        (dolist (w2 (cdr (member w twords)))
          (format out "SIM ~S ~S ~,3F~\%"
                  w w2 (similarity w w2)))))))
```

Bibliography

Anderson, J. R. (1983). A spreading activation theory of memory. *Journal of Verbal Learning and Verbal Behaviour*, 22.

Balota, D. A. and Lorch Jr., R. F. (1986). Depth of automatic spreading activation: Mediated priming effects in pronounciation but not in lexical decision. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 12(3):336–345.

Banaji, M. R. and Greenwald, A. G. (1994). Implicit stereotyping and prejudice.

Beer, A. L. and Diehl, V. A. (2001). The role of short-term memory in semantic priming. *The Journal of General Psychology*, 128(3):329–350.

Boroditsky, L. and Ramscar, M. (2002). The roles of body and mind in abstract thought. *Psychological Science*, 13(2):185–188.

Church, K. and Hanks, P. (1989). Word association norms, mutual information and lexicography. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*. Vancouver, British Columbia, Canada: Association for Computational Linguistics.

de Groot, A. M. B. (1983). The range of automatic spreading activation in word priming. *Journal of Verbal Learning and Verbal Behaviour*, pages 417–436.

Deerwester, S., Dumais, S., Landauer, T., Furnas, G., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the Society for Information Science*, 41(6):391–407.

Dunn, E. W., Moore, M., and Nosek, B. A. (2005). The war of the words: How linguistic differences in reporting shape perceptions of terrorism. *Analyses of Social Issues and Public Policy*, 5(1):67–86.

Extreme-Programming (1999). http://www.extremeprogramming.org/.

Finch, S. (1993). *Finding structure in language*. PhD thesis, University of Edinburgh.

Fodor, J. and Lepor, E. (1999). All at sea in semantic space: Churchland on meaning similarity. *the Journal of Philosophy*, 96:318–403.

French, R. M. (2000). Peeking behind the screen: The unsuspected power of the standard turing test. *Journal of Experimental and Theoretical Artificial Intelligence*, 12:331–340.

French, R. M. and Labiouse, C. (2002). Four problems with extracting human semantics from large text corpora. In *Proceedings of the 24th Annual Conference of the Cognitive Science Society*.

Greenwald, A. G., McGhee, D. E., and Schwartz, J. L. K. (1998). Measuring individual differences in implicit cognition: The implicit association test. *Journal of Personality and Social Psychology*, 74(6):1464–1480.

Greenwald, A. G. and Nosek, B. A. (2001). Health of the implicit association test at age 3. *Zeitschrift fur Experimentelle Psychologie*, 48:85–93.

Implicit-Project (1998). https://implicit.harvard.edu/implicit/.

Lamkins, D. B. (2004). *Successful Lisp: How to Understand and Use Common Lisp*. bookfix.com.

Landauer, T. (2002). Applications of latent semantic analysis. In *24th Annual Meeting of the Cognitive Science Society*.

Landauer, T. and Dumais, S. (1997). A solution to plato's problem: The latent semantic analysis theory of acquisition, induction and representation of knowledge. *Psychological Review*, 104:211–240.

Landauer, T., Foltz, P., and Laham, D. (1998). Introduction to latent semantic analysis. *Discourse Process*, 25:259–284.

LAPACK (2000). http://www.netlib.org/lapack/.

Lemm, K. and Banaji, M. R. (1999). Unconcious attitudes and beliefs about women and men. In Pasero, U. and Braun, F., editors, *Wahrnehmung und Herstellung von Geschlecht (Perceiving and performing gender)*, pages 215–233. Opladen: Westdutscher Verlag.

Levy, J. P. and Bullinaria, J. A. (2001). *Learning lexical properties from word usage patterns*, pages 273–282. Springer-Verlang.

Levy, J. P., Bullinaria, J. A., and Patel, M. (1998). Explorations in the derivations of word co-occurrence statistics. *South Pacific Journal of Psychology*, 10(1):99–111.

Lowe, W. (2000). *Topographic Maps of Semantic Space*. PhD thesis, Institute for Adaptive and Neural Computation, Edinburgh University.

Lowe, W. (2001). Towards a theory of semantic space. In Moore, J. D. and Stenning, K., editors, *Proceedings of the Twenty-Third Annual Conference of the Cognitive Science Society*, pages 576–581, Mahwah NJ. Lawrence Erlbaum Associates.

Lowe, W. and McDonald, S. (2000). The direct route: Mediated priming in semantic space. In Gleitman, L. R. and Joshi, J. K., editors, *Proceedings of the Twenty-Second Annual Conference of the Cognitive Science Society*, pages 806–811, Mahwah NJ. Lawrence Erlbaum Associates.

Mitchell, J. P., Nosek, B. A., and Banaji, M. R. (2003). Contextual variations in implicit evaluation. *Journal of Experimental Psychology: General*, 132(3):455–469.

Patel, M., Bullinaria, J. A., and Levy, J. P. (1997). Extracting semantic representations from large text corpora. In *Proceedings of the Fourth Neural Computation and PsychologyWorkshop*, pages 199–212, London. Springer-Verlag.

Piaget, J. (1950). *The psychology of intelligence*. London: Routledge and Kagan Paul.

Project-Gutenberg-Literary-Archive-Foundation (2006). http://www.gutenberg.org.

Ratcliff, R. and McKoon, G. (1988). A retrieval theory of priming in memory. *Psychological Review*.

Ratcliff, R. and McKoon, G. (1992). Spreading activation versus compound cue accounts of priming: Mediated priming revisited. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 18(6):1155–1172.

Ratcliff, R. and McKoon, G. (1995). Sequential effects in lexical decisions: Tests of compound cue retrieval theory. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 21:1380–1388.

Sharifian, F. and Samani, R. (1997). Hierarchical spreading of activation. In Sharifian, F., editor, *Proceedings of the Conference on Language, Cognition and Interpretation*, pages 1–10. Isfahan: IAU Press.

Steele Jr., G. L. (1990). *Common Lisp The Language*. Digital Press, 2 edition.

Steele Jr., G. L. and Gabriel, R. P. (1993). The evolution of lisp. *ACM SIGPLAN Notices*, 28(3).

Waters, R. C. (1991). Some useful lisp algorithms: Part 1. Technical Report 91-04, Mitsubishi Electric Research Laboratories.

Zipf, G. K. (1949). *Human Behaviour and the Principle of Least-Effort*. Addison Wsley, Cambridge MA.